

Let's Build A Web Server. Part 1.

(<https://ruslanspivak.com/lbaws-part1/>)

Date  Mon, March 09, 2015

Out for a walk one day, a woman came across a construction site and saw three men working. She asked the first man, "What are you doing?" Annoyed by the question, the first man barked, "Can't you see that I'm laying bricks?" Not satisfied with the answer, she asked the second man what he was doing. The second man answered, "I'm building a brick wall." Then, turning his attention to the first man, he said, "Hey, you just passed the end of the wall. You need to take off that last brick." Again not satisfied with the answer, she asked the third man what he was doing. And the man said to her while looking up in the sky, "I am building the biggest cathedral this world has ever known." While he was standing there and looking up in the sky the other two men started arguing about the errant brick. The man turned to the first two men and said, "Hey guys, don't worry about that brick. It's an inside wall, it will get plastered over and no one will ever see that brick. Just move on to another layer."¹

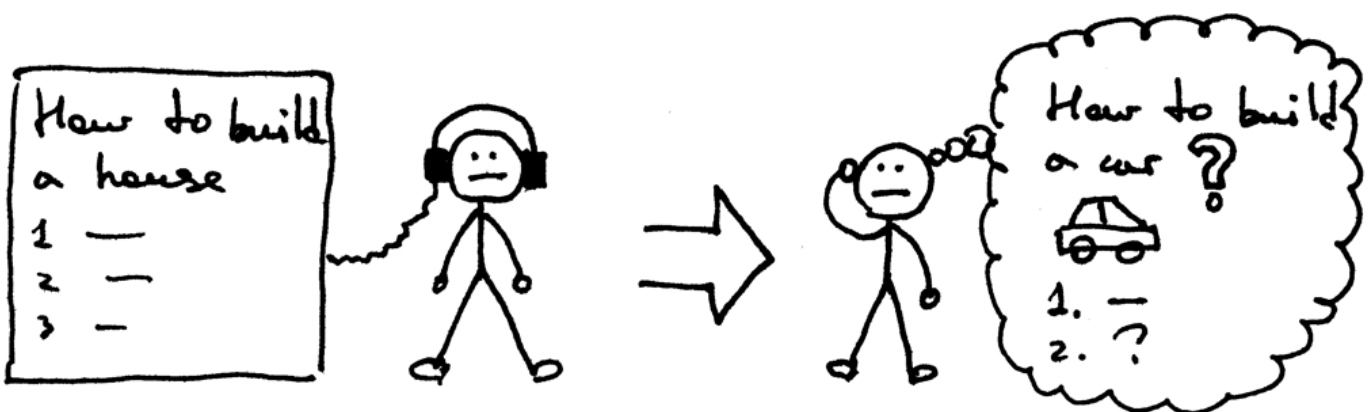
The moral of the story is that when you know the whole system and understand how different pieces fit together (bricks, walls, cathedral), you can identify and fix problems faster (errant brick).

What does it have to do with creating your own Web server from scratch?

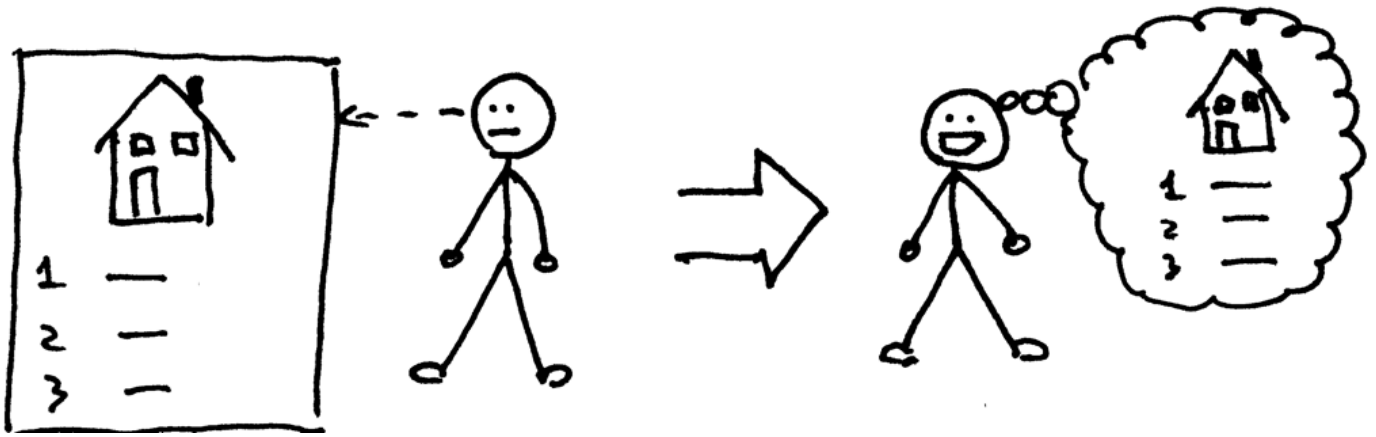
I believe to become a better developer you MUST get a better understanding of the underlying software systems you use on a daily basis and that includes programming languages, compilers and interpreters, databases and operating systems, web servers and web frameworks. And, to get a better and deeper understanding of those systems you MUST re-build them from scratch, brick by brick, wall by wall.

Confucius put it this way:

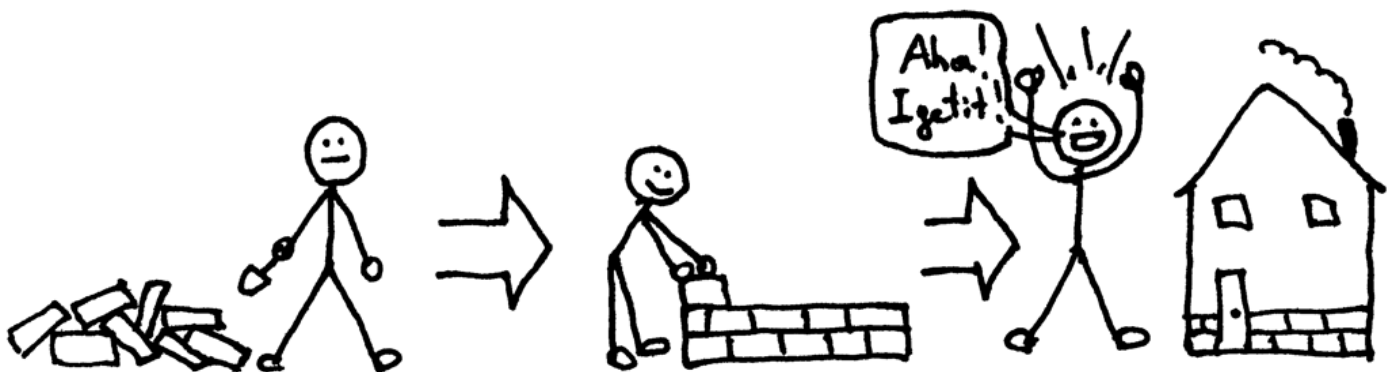
"I hear and I forget."



"I see and I remember."



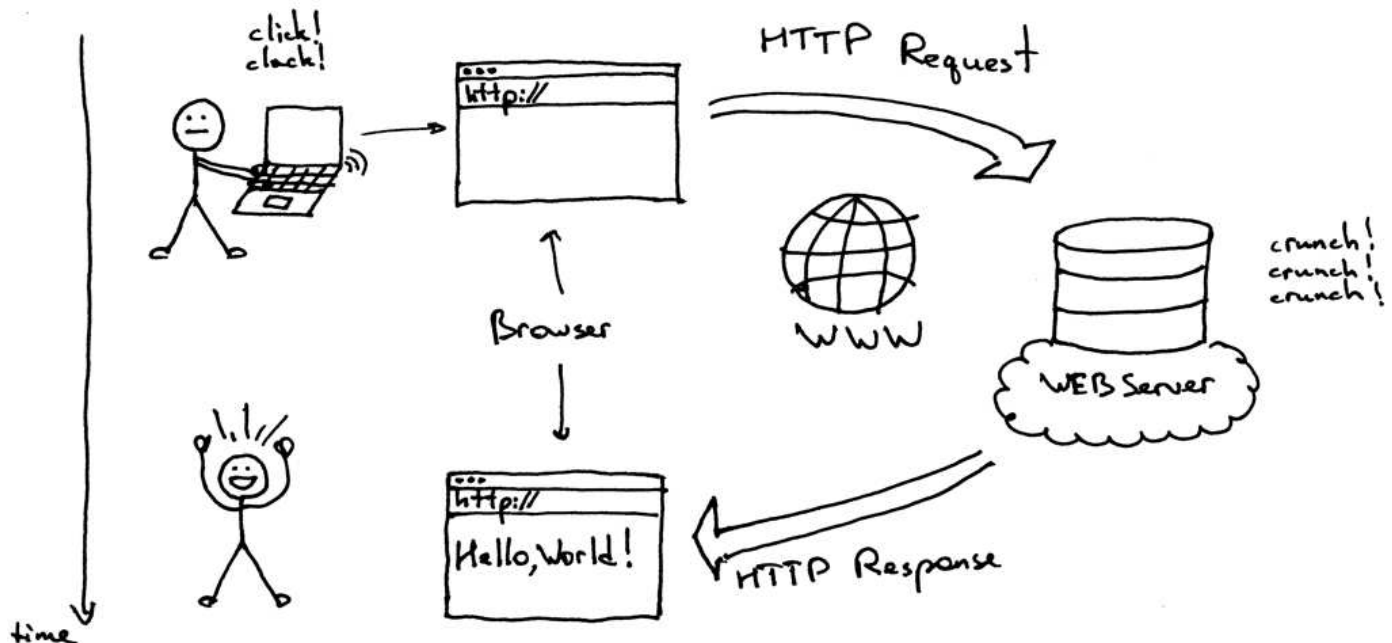
"I do and I understand."



I hope at this point you're convinced that it's a good idea to start re-building different software systems to learn how they work.

In this three-part series I will show you how to build your own basic Web server. Let's get started.

First things first, what is a Web server?



In a nutshell it's a networking server that sits on a physical server (oops, a server on a server) and waits for a client to send a request. When it receives a request, it generates a response and sends it back to the client. The communication between a client and a server happens using HTTP protocol. A client can be your browser or any other software that speaks HTTP.

What would a very simple implementation of a Web server look like? Here is my take on it. The example is in Python but even if you don't know Python (it's a very easy language to pick up, try it!) you still should be able to understand concepts from the code and explanations below:

```
import socket

HOST, PORT = '', 8888

listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
listen_socket.bind((HOST, PORT))
listen_socket.listen(1)
print 'Serving HTTP on port %s ...' % PORT
while True:
    client_connection, client_address = listen_socket.accept()
    request = client_connection.recv(1024)
    print request

    http_response = """\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)
    client_connection.close()
```

Save the above code as `webserver1.py` or download it directly from [GitHub](https://github.com/rspivak/lbaws/blob/master/part1/webserver1.py) (<https://github.com/rspivak/lbaws/blob/master/part1/webserver1.py>) and run it on the command line like this

```
$ python webserver1.py
Serving HTTP on port 8888 ...
```

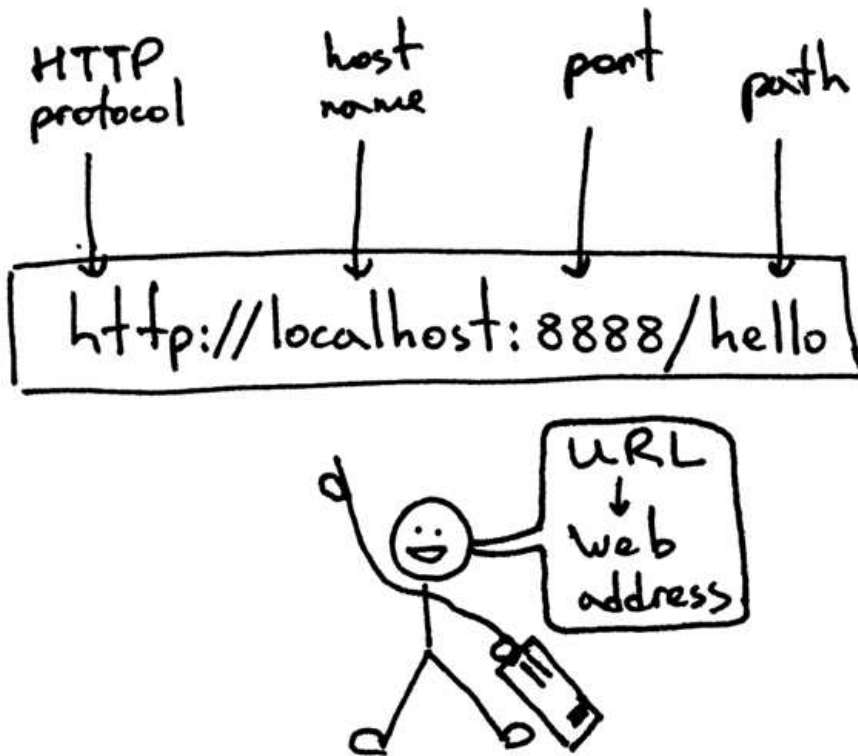
Now type in the following URL in your Web browser's address bar <http://localhost:8888/hello> (<http://localhost:8888/hello>), hit Enter, and see magic in action. You should see *"Hello, World!"* displayed in your browser like this:



Just do it, seriously. I will wait for you while you're testing it.

Done? Great. Now let's discuss how it all actually works.

First let's start with the Web address you've entered. It's called an URL (http://en.wikipedia.org/wiki/Uniform_resource_locator) and here is its basic structure:



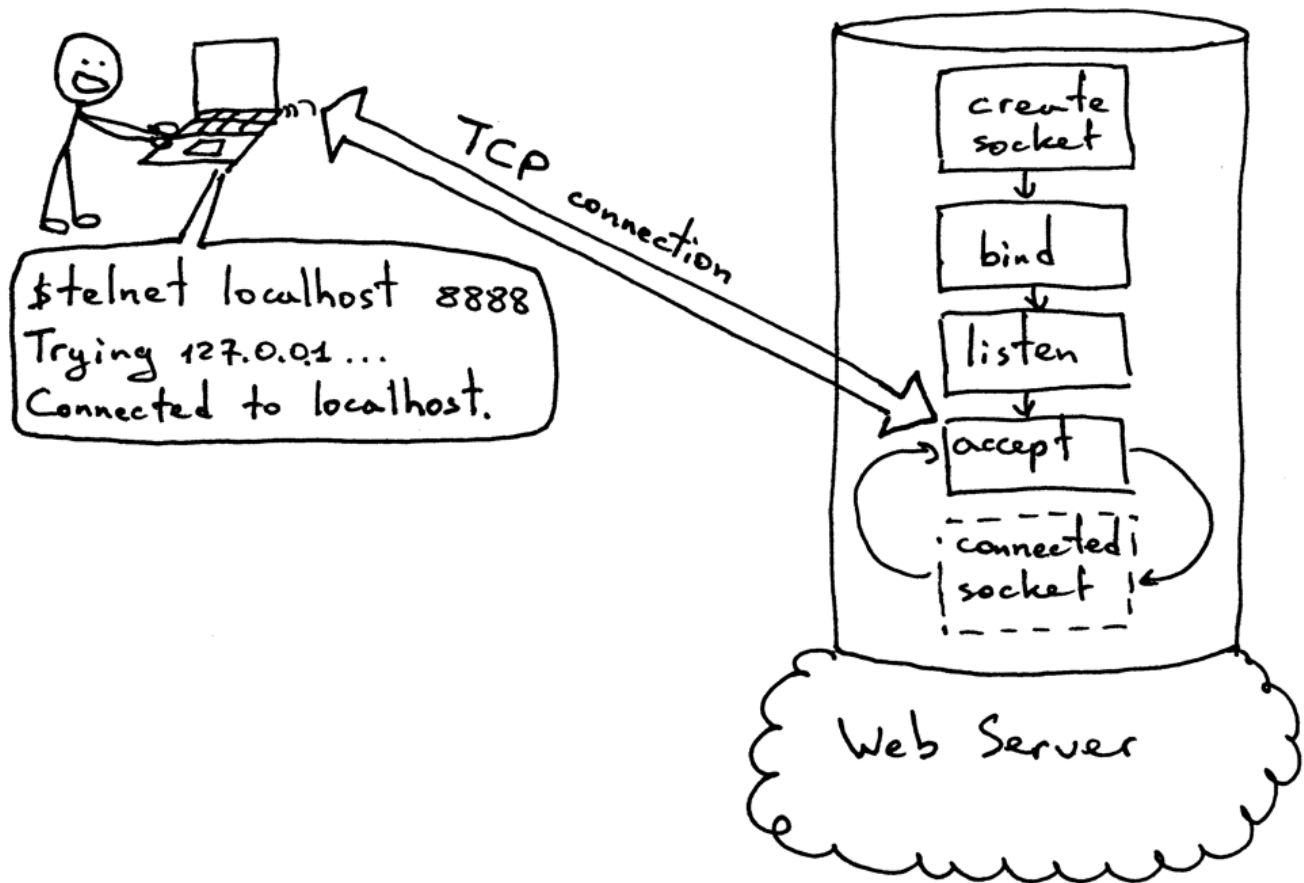
This is how you tell your browser the address of the Web server it needs to find and connect to and the page (path) on the server to fetch for you. Before your browser can send a HTTP request though, it first needs to establish a TCP connection with the Web server. Then it sends an HTTP request over the TCP connection to the server and waits for the server to send an HTTP response back. And when your browser receives the response it displays it, in this case it displays "Hello, World!"

Let's explore in more detail how the client and the server establish a TCP connection before sending HTTP requests and responses. To do that they both use so-called *sockets*. Instead of using a browser directly you are going to simulate your browser manually by using *telnet* on the command line.

On the same computer you're running the Web server fire up a telnet session on the command line specifying a host to connect to *localhost* and the port to connect to *8888* and then press Enter:

```
$ telnet localhost 8888
Trying 127.0.0.1 ...
Connected to localhost.
```

At this point you've established a TCP connection with the server running on your local host and ready to send and receive HTTP messages. In the picture below you can see a standard procedure a server has to go through to be able to accept new TCP connections.

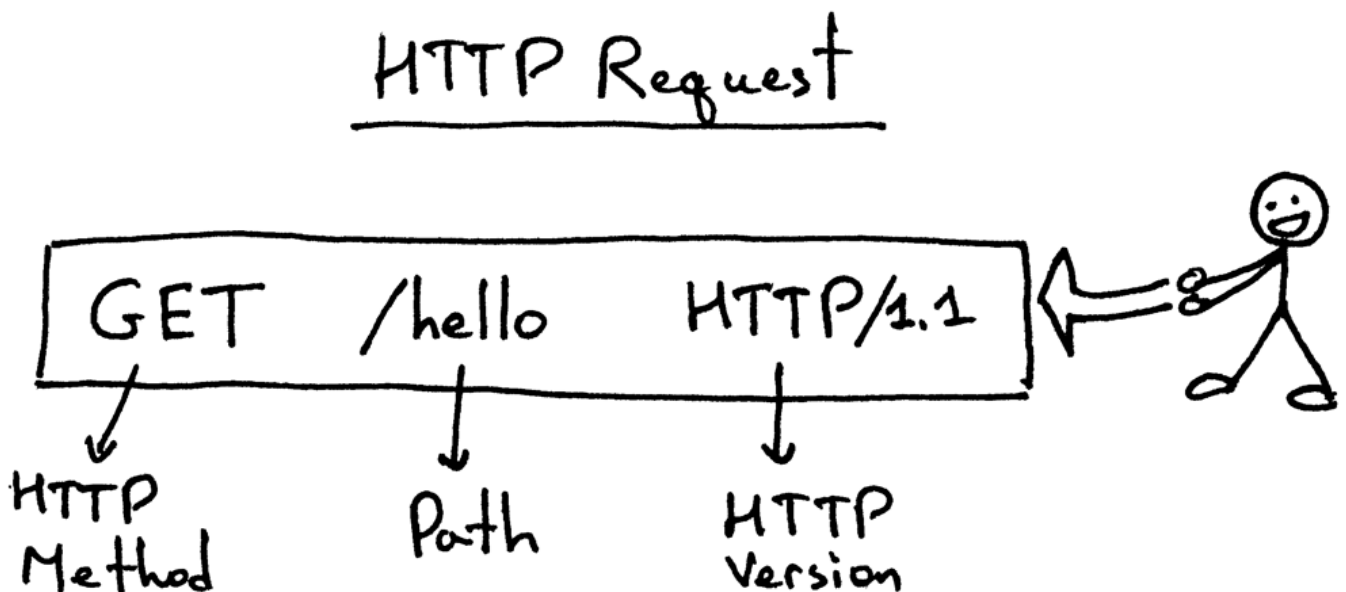


In the same telnet session type **GET /hello HTTP/1.1** and hit Enter:

```
$ telnet localhost 8888
Trying 127.0.0.1 ...
Connected to localhost.
GET /hello HTTP/1.1

HTTP/1.1 200 OK
Hello, World!
```

You've just manually simulated your browser! You sent an HTTP request and got an HTTP response back. This is the basic structure of an HTTP request:

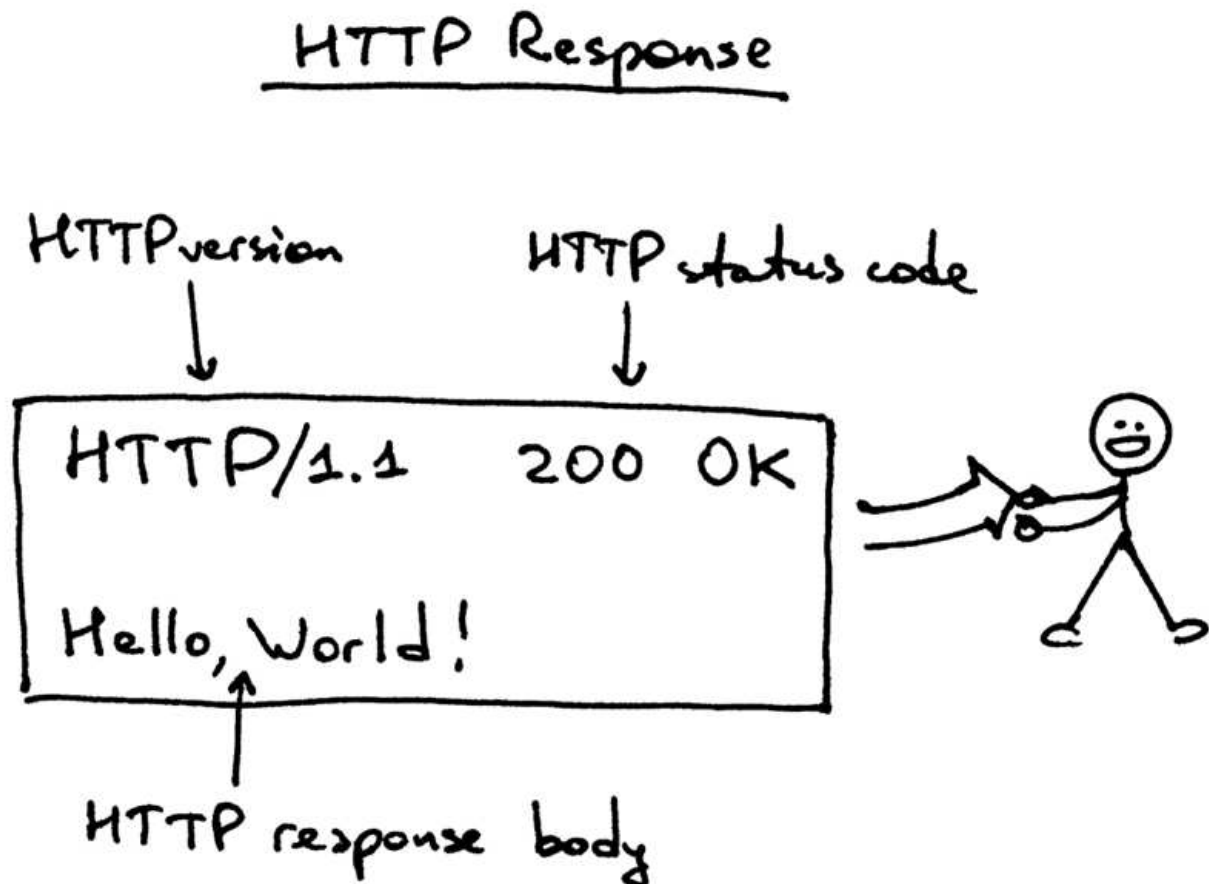


The HTTP request consists of the line indicating the HTTP method (**GET**, because we are asking our server to return us something), the path `/hello` that indicates a “page” on the server we want and the protocol version.

For simplicity's sake our Web server at this point completely ignores the above request line. You could just as well type in any garbage instead of “GET /hello HTTP/1.1” and you would still get back a “Hello, World!” response.

Once you've typed the request line and hit Enter the client sends the request to the server, the server reads the request line, prints it and returns the proper HTTP response.

Here is the HTTP response that the server sends back to your client (*telnet* in this case):



Let's dissect it. The response consists of a status line `HTTP/1.1 200 OK`, followed by a required empty line, and then the HTTP response body.

The response status line `HTTP/1.1 200 OK` consists of the *HTTP Version*, the *HTTP status code* and the *HTTP status code reason phrase* `OK`. When the browser gets the response, it displays the body of the response and that's why you see “Hello, World!” in your browser.

And that's the basic model of how a Web server works. To sum it up: The Web server creates a listening socket and starts accepting new connections in a loop. The client initiates a TCP connection and, after successfully establishing it, the client sends an HTTP request to the server and the server responds with an HTTP response that gets displayed to the user. To establish a TCP connection both clients and servers use *sockets*.

Now you have a very basic working Web server that you can test with your browser or some other HTTP client. As you've seen and hopefully tried, you can also be a human HTTP client too, by using *telnet* and typing HTTP requests manually.

Here's a question for you: "How do you run a Django application, Flask application, and Pyramid application under your freshly minted Web server without making a single change to the server to accommodate all those different Web frameworks?"

I will show you exactly how in Part 2 of the series. Stay tuned.

BTW, I'm writing a book **"Let's Build A Web Server: First Steps"** that explains how to write a basic web server from scratch and goes into more detail on topics I just covered. Subscribe to the mailing list to get the latest updates about the book and the release date.

Enter Your First Name *

Enter Your Best Email *

Get Updates!

All articles in this series:

- [Let's Build A Web Server. Part 1. \(http://ruslanspivak.com/lbaws-part1/\)](http://ruslanspivak.com/lbaws-part1/)
- [Let's Build A Web Server. Part 2. \(http://ruslanspivak.com/lbaws-part2/\)](http://ruslanspivak.com/lbaws-part2/)
- [Let's Build A Web Server. Part 3. \(http://ruslanspivak.com/lbaws-part3/\)](http://ruslanspivak.com/lbaws-part3/)

1. Inspired by [Lead with a Story: A Guide to Crafting Business Narratives That Captivate, Convince, and Inspire \(http://www.amazon.com/gp/product/0814420303/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0814420303&linkCode=as2&tag=russblo0b-20&linkId=HY2LNXTSGPPFZ2EV\)](http://www.amazon.com/gp/product/0814420303/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0814420303&linkCode=as2&tag=russblo0b-20&linkId=HY2LNXTSGPPFZ2EV)

Comments

78 Comments

Ruslan's Blog

Login ▾

Recommend 36

Tweet

Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name



ChinaMoe • 4 years ago

how a great post!

77 ^ | ▾ • Reply • Share ›



LynnRice1 • 4 years ago

Last night I created an account here and left a message asking for help because the first exercise failed. Only difference I could see was that I was using python 3.4. (yes I Changed the print statements). Anyway, My message that was here last night is not showing up. So, maybe this one will. help


3 ^ | ▾ • Reply • Share ›



Charles MacKay → LynnRice1 • 2 years ago

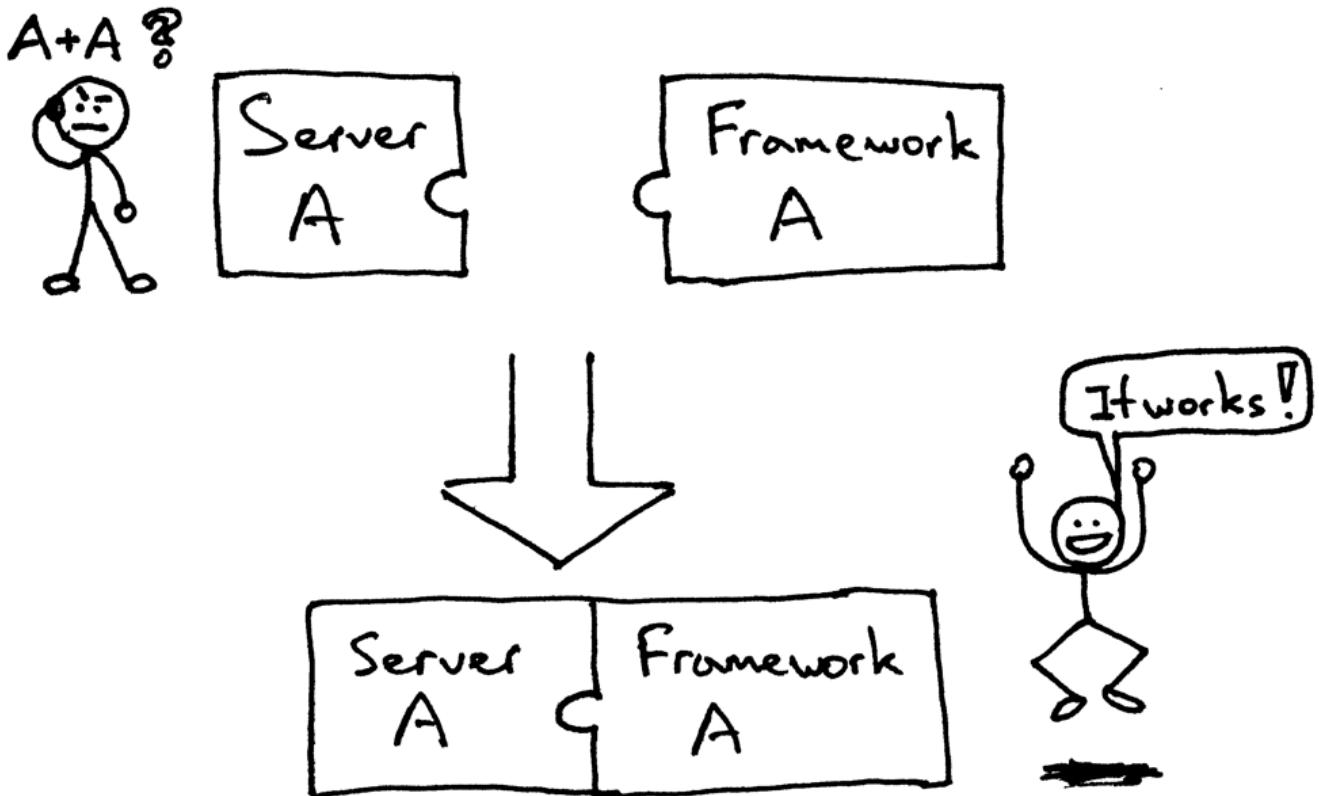
Let's Build A Web Server. Part 2.

(<https://ruslanspivak.com/lbaws-part2/>)

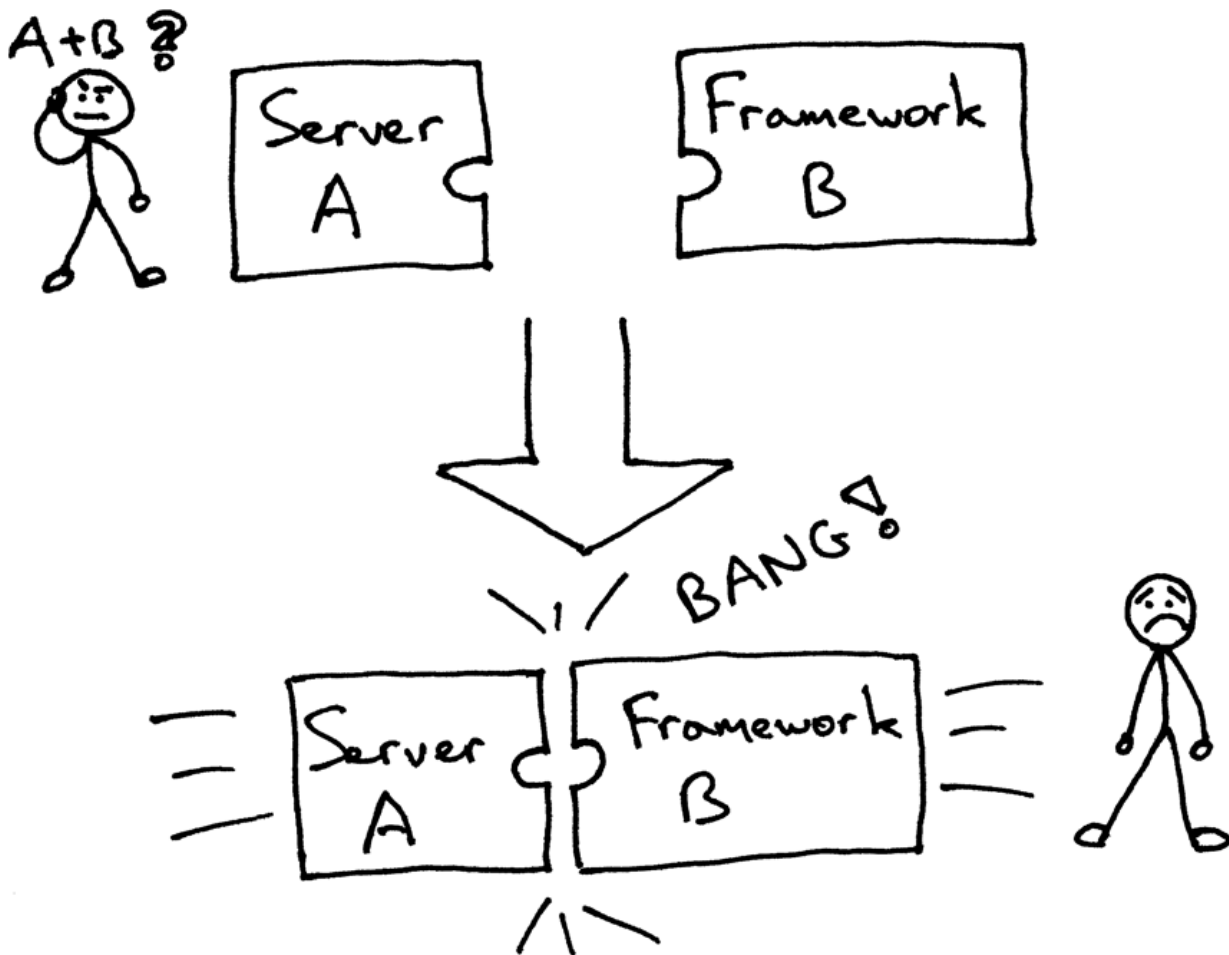
Date  Mon, April 06, 2015

Remember, in [Part 1 \(http://ruslanspivak.com/lbaws-part1/\)](http://ruslanspivak.com/lbaws-part1/) I asked you a question: “How do you run a Django application, Flask application, and Pyramid application under your freshly minted Web server without making a single change to the server to accommodate all those different Web frameworks?” Read on to find out the answer.

In the past, your choice of a Python Web framework would limit your choice of usable Web servers, and vice versa. If the framework and the server were designed to work together, then you were okay:



But you could have been faced (and maybe you were) with the following problem when trying to combine a server and a framework that weren't designed to work together:

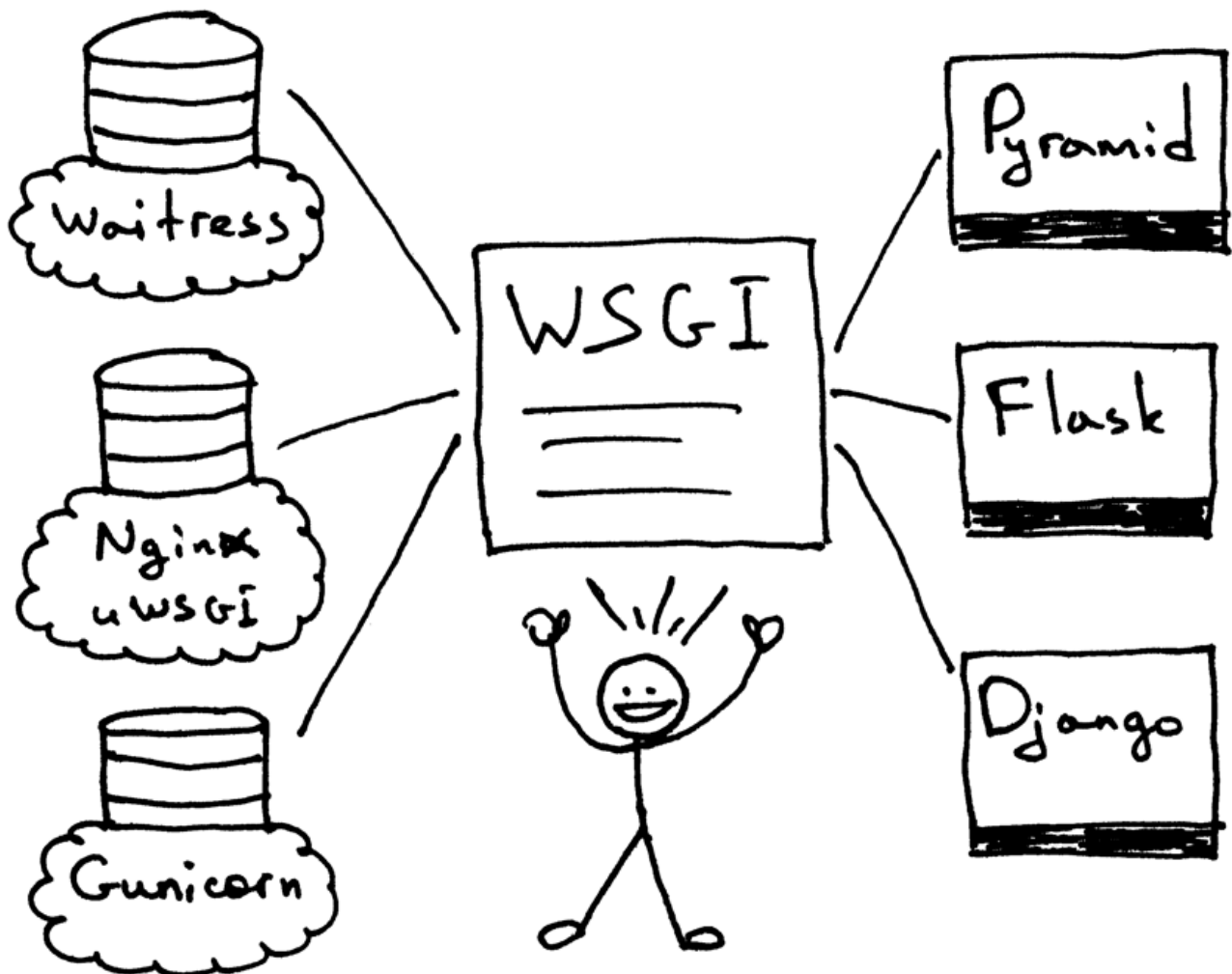


Basically you had to use what worked together and not what you might have wanted to use.

So, how do you then make sure that you can run your Web server with multiple Web frameworks without making code changes either to the Web server or to the Web frameworks? And the answer to that problem became the **Python Web Server Gateway Interface** (or WSGI (<https://www.python.org/dev/peps/pep-0333/>) for short, pronounced “wizgy”).



WSGI (<https://www.python.org/dev/peps/pep-0333/>) allowed developers to separate choice of a Web framework from choice of a Web server. Now you can actually mix and match Web servers and Web frameworks and choose a pairing that suits your needs. You can run Django (<https://www.djangoproject.com/>), Flask (<http://flask.pocoo.org/>), or Pyramid (<http://trypyramid.com/>), for example, with Gunicorn (<http://gunicorn.org/>) or Nginx/uWSGI (<http://uwsgi-docs.readthedocs.org>) or Waitress (<http://waitress.readthedocs.org>). Real mix and match, thanks to the WSGI support in both servers and frameworks:



So, WSGI (<https://www.python.org/dev/peps/pep-0333/>) is the answer to the question I asked you in Part 1 (<http://ruslanspivak.com/lbaws-part1/>) and repeated at the beginning of this article. Your Web server must implement the server portion of a WSGI interface and all modern Python Web Frameworks already implement the framework side of the WSGI interface, which allows you to use them with your Web server without ever modifying your server's code to accommodate a particular Web framework.

Now you know that WSGI support by Web servers and Web frameworks allows you to choose a pairing that suits you, but it is also beneficial to server and framework developers because they can focus on their preferred area of specialization and not step on each other's toes. Other languages have similar interfaces too: Java, for example, has Servlet API (http://en.wikipedia.org/wiki/Java_servlet) and Ruby has Rack (http://en.wikipedia.org/wiki/Rack_%28web_server_interface%29).

It's all good, but I bet you are saying: "Show me the code!" Okay, take a look at this pretty minimalistic WSGI server implementation:

```
# Tested with Python 2.7.9, Linux & Mac OS X
import socket
import StringIO
import sys

class WSGIServer(object):

    address_family = socket.AF_INET
    socket_type = socket.SOCK_STREAM
    request_queue_size = 1

    def __init__(self, server_address):
        # Create a Listening socket
        self.listen_socket = listen_socket = socket.socket(
            self.address_family,
            self.socket_type
        )
        # Allow to reuse the same address
        listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        # Bind
        listen_socket.bind(server_address)
        # Activate
        listen_socket.listen(self.request_queue_size)
        # Get server host name and port
        host, port = self.listen_socket.getsockname()[:2]
        self.server_name = socket.getfqdn(host)
        self.server_port = port
        # Return headers set by Web framework/Web application
        self.headers_set = []

    def set_app(self, application):
        self.application = application

    def serve_forever(self):
        listen_socket = self.listen_socket
        while True:
            # New client connection
            self.client_connection, client_address = listen_socket.accept()
            # Handle one request and close the client connection. Then
            # loop over to wait for another client connection
            self.handle_one_request()

    def handle_one_request(self):
        self.request_data = request_data = self.client_connection.recv(1024)
        # Print formatted request data a la 'curl -v'
        print(''.join(
            '< {line}\n'.format(line=line)
            for line in request_data.splitlines()
        ))

        self.parse_request(request_data)

        # Construct environment dictionary using request data
        env = self.get_environ()
```

```

# It's time to call our application callable and get
# back a result that will become HTTP response body
result = self.application(env, self.start_response)

# Construct a response and send it back to the client
self.finish_response(result)

def parse_request(self, text):
    request_line = text.splitlines()[0]
    request_line = request_line.rstrip('\r\n')
    # Break down the request line into components
    (self.request_method, # GET
     self.path, # /hello
     self.request_version # HTTP/1.1
     ) = request_line.split()

def get_environ(self):
    env = {}
    # The following code snippet does not follow PEP8 conventions
    # but it's formatted the way it is for demonstration purposes
    # to emphasize the required variables and their values
    #
    # Required WSGI variables
    env['wsgi.version'] = (1, 0)
    env['wsgi.url_scheme'] = 'http'
    env['wsgi.input'] = StringIO.StringIO(self.request_data)
    env['wsgi.errors'] = sys.stderr
    env['wsgi.multithread'] = False
    env['wsgi.multiprocess'] = False
    env['wsgi.run_once'] = False
    # Required CGI variables
    env['REQUEST_METHOD'] = self.request_method # GET
    env['PATH_INFO'] = self.path # /hello
    env['SERVER_NAME'] = self.server_name # localhost
    env['SERVER_PORT'] = str(self.server_port) # 8888
    return env

def start_response(self, status, response_headers, exc_info=None):
    # Add necessary server headers
    server_headers = [
        ('Date', 'Tue, 31 Mar 2015 12:54:48 GMT'),
        ('Server', 'WSGIServer 0.2'),
    ]
    self.headers_set = [status, response_headers + server_headers]
    # To adhere to WSGI specification the start_response must return
    # a 'write' callable. We simplicity's sake we'll ignore that detail
    # for now.
    # return self.finish_response

def finish_response(self, result):
    try:
        status, response_headers = self.headers_set
        response = 'HTTP/1.1 {status}\r\n'.format(status=status)
        for header in response_headers:
            response += '{0}: {1}\r\n'.format(*header)
        response += '\r\n'

```

```

        for data in result:
            response += data
        # Print formatted response data a La 'curl -v'
        print(''.join(
            '> {line}\n'.format(line=line)
            for line in response.splitlines()
        ))
        self.client_connection.sendall(response)
    finally:
        self.client_connection.close()

```

```
SERVER_ADDRESS = (HOST, PORT) = '', 8888
```

```

def make_server(server_address, application):
    server = WSGIServer(server_address)
    server.set_app(application)
    return server

```

```

if __name__ == '__main__':
    if len(sys.argv) < 2:
        sys.exit('Provide a WSGI application object as module:callable')
    app_path = sys.argv[1]
    module, application = app_path.split(':')
    module = __import__(module)
    application = getattr(module, application)
    httpd = make_server(SERVER_ADDRESS, application)
    print('WSGIServer: Serving HTTP on port {port} ...\n'.format(port=PORT))
    httpd.serve_forever()

```

It's definitely bigger than the server code in [Part 1 \(http://ruslanspivak.com/lbaws-part1/\)](http://ruslanspivak.com/lbaws-part1/), but it's also small enough (just under 150 lines) for you to understand without getting bogged down in details. The above server also does more - it can run your basic Web application written with your beloved Web framework, be it Pyramid, Flask, Django, or some other Python WSGI framework.

Don't believe me? Try it and see for yourself. Save the above code as `webserver2.py` or download it directly from [GitHub \(https://github.com/rspivak/lbaws/blob/master/part2/webserver2.py\)](https://github.com/rspivak/lbaws/blob/master/part2/webserver2.py). If you try to run it without any parameters it's going to complain and exit.

```

$ python webserver2.py
Provide a WSGI application object as module:callable

```

It really wants to serve your Web application and that's where the fun begins. To run the server the only thing you need installed is Python. But to run applications written with Pyramid, Flask, and Django you need to install those frameworks first. Let's install all three of them. My preferred method is by using [virtualenv \(https://virtualenv.pypa.io\)](https://virtualenv.pypa.io). Just follow the steps below to create and activate a virtual environment and then install all three Web frameworks.

```
$ [sudo] pip install virtualenv
$ mkdir ~/envs
$ virtualenv ~/envs/lbaws/
$ cd ~/envs/lbaws/
$ ls
bin  include  lib
$ source bin/activate
(lbaws) $ pip install pyramid
(lbaws) $ pip install flask
(lbaws) $ pip install django
```

At this point you need to create a Web application. Let's start with Pyramid (<http://trypyramid.com/>) first. Save the following code as *pyramidapp.py* to the same directory where you saved *webserver2.py* or download the file directly from GitHub (<https://github.com/rspivak/lbaws/blob/master/part2/pyramidapp.py>):

```
from pyramid.config import Configurator
from pyramid.response import Response

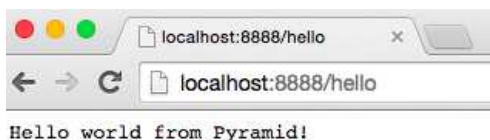
def hello_world(request):
    return Response(
        'Hello world from Pyramid!\n',
        content_type='text/plain',
    )

config = Configurator()
config.add_route('hello', '/hello')
config.add_view(hello_world, route_name='hello')
app = config.make_wsgi_app()
```

Now you're ready to serve your Pyramid application with your very own Web server:

```
(lbaws) $ python webserver2.py pyramidapp:app
WSGIServer: Serving HTTP on port 8888 ...
```

You just told your server to load the *'app'* callable from the python module *'pyramidapp'* Your server is now ready to take requests and forward them to your Pyramid application. The application only handles one route now: the */hello* route. Type <http://localhost:8888/hello> (<http://localhost:8888/hello>) address into your browser, press Enter, and observe the result:



You can also test the server on the command line using the *'curl'* utility:

```
$ curl -v http://localhost:8888/hello
...
```

Check what the server and *curl* prints to standard output.

Now onto Flask (<http://flask.pocoo.org/>). Let's follow the same steps.

```
from flask import Flask
from flask import Response
flask_app = Flask('flaskapp')

@flask_app.route('/hello')
def hello_world():
    return Response(
        'Hello world from Flask!\n',
        mimetype='text/plain'
    )

app = flask_app.wsgi_app
```

Save the above code as *flaskapp.py* or download it from [GitHub](https://github.com/rspivak/lbaws/blob/master/part2/flaskapp.py) (<https://github.com/rspivak/lbaws/blob/master/part2/flaskapp.py>) and run the server as:

```
(lbaws) $ python webserver2.py flaskapp:app
WSGIServer: Serving HTTP on port 8888 ...
```

Now type in the <http://localhost:8888/hello> (<http://localhost:8888/hello>) into your browser and press Enter:



Again, try *'curl'* and see for yourself that the server returns a message generated by the Flask application:

```
$ curl -v http://localhost:8888/hello
...
```

Can the server also handle a [Django](https://www.djangoproject.com/) (<https://www.djangoproject.com/>) application? Try it out! It's a little bit more involved, though, and I would recommend cloning the whole repo and use [djangoapp.py](https://github.com/rspivak/lbaws/blob/master/part2/djangoapp.py) (<https://github.com/rspivak/lbaws/blob/master/part2/djangoapp.py>), which is part of the [GitHub repository](https://github.com/rspivak/lbaws/) (<https://github.com/rspivak/lbaws/>). Here is the source code which basically adds the Django *'helloworld'* project (pre-created using Django's *django-admin.py startproject* command) to the current Python path and then imports the project's WSGI application.

```
import sys
sys.path.insert(0, './helloworld')
from helloworld import wsgi

app = wsgi.application
```

Save the above code as *djangoapp.py* and run the Django application with your Web server:

```
(lbaws) $ python webserver2.py djangoapp:app
WSGIServer: Serving HTTP on port 8888 ...
```

Type in the following address and press Enter:



And as you've already done a couple of times before, you can test it on the command line, too, and confirm that it's the Django application that handles your requests this time around:

```
$ curl -v http://localhost:8888/hello
...
```

Did you try it? Did you make sure the server works with those three frameworks? If not, then please do so. Reading is important, but this series is about rebuilding and that means you need to get your hands dirty. Go and try it. I will wait for you, don't worry. No seriously, you must try it and, better yet, retype everything yourself and make sure that it works as expected.

Okay, you've experienced the power of WSGI: it allows you to mix and match your Web servers and Web frameworks. WSGI provides a minimal interface between Python Web servers and Python Web Frameworks. It's very simple and it's easy to implement on both the server and the framework side. The following code snippet shows the server and the framework side of the interface:

```
def run_application(application):
    """Server code."""
    # This is where an application/framework stores
    # an HTTP status and HTTP response headers for the server
    # to transmit to the client
    headers_set = []
    # Environment dictionary with WSGI/CGI variables
    environ = {}

    def start_response(status, response_headers, exc_info=None):
        headers_set[:] = [status, response_headers]

    # Server invokes the 'application' callable and gets back the
    # response body
    result = application(environ, start_response)
    # Server builds an HTTP response and transmits it to the client
    ...

def app(environ, start_response):
    """A barebones WSGI app."""
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Hello world!']

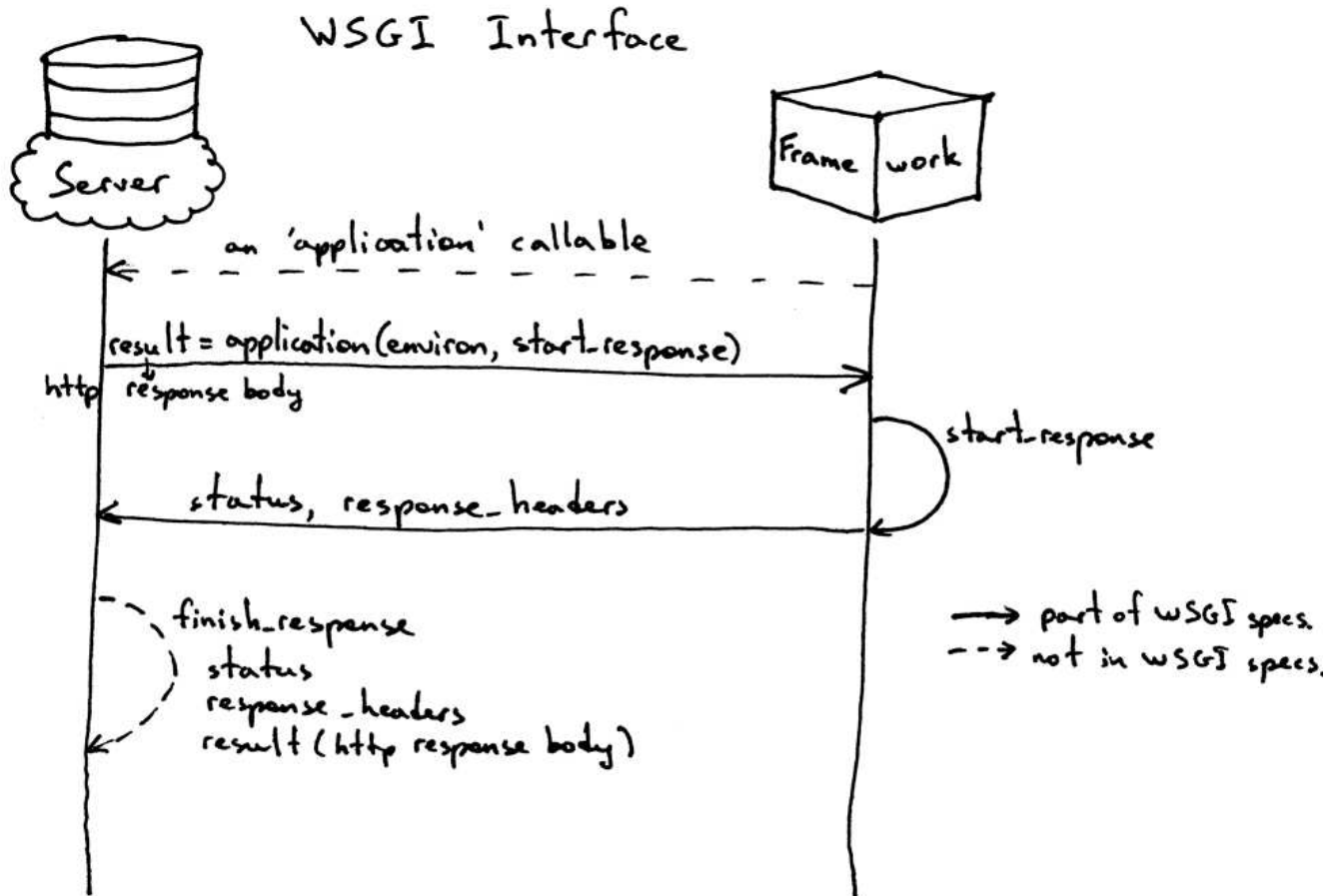
run_application(app)
```

Here is how it works:

1. The framework provides an *'application'* callable (The WSGI specification doesn't prescribe how that should be implemented)
2. The server invokes the *'application'* callable for each request it receives from an HTTP client. It passes a dictionary *'environ'* containing WSGI/CGI variables and a *'start_response'* callable as arguments to the *'application'* callable.

3. The framework/application generates an HTTP status and HTTP response headers and passes them to the `'start_response'` callable for the server to store them. The framework/application also returns a response body.
4. The server combines the status, the response headers, and the response body into an HTTP response and transmits it to the client (This step is not part of the specification but it's the next logical step in the flow and I added it for clarity)

And here is a visual representation of the interface:



So far, you've seen the Pyramid, Flask, and Django Web applications and you've seen the server code that implements the server side of the WSGI specification. You've even seen the barebones WSGI application code snippet that doesn't use any framework.

The thing is that when you write a Web application using one of those frameworks you work at a higher level and don't work with WSGI directly, but I know you're curious about the framework side of the WSGI interface, too because you're reading this article. So, let's create a minimalistic WSGI Web application/Web framework without using Pyramid, Flask, or Django and run it with your server:

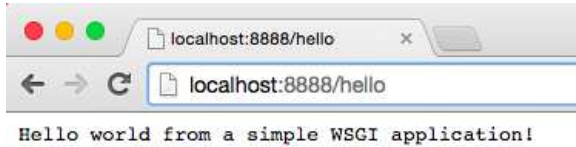
```
def app(environ, start_response):
    """A barebones WSGI application.

    This is a starting point for your own Web framework :)
    """
    status = '200 OK'
    response_headers = [('Content-Type', 'text/plain')]
    start_response(status, response_headers)
    return ['Hello world from a simple WSGI application!\n']
```

Again, save the above code in `wsgiapp.py` file or download it from [GitHub](https://github.com/rspivak/lbaws/blob/master/part2/wsgiapp.py) (<https://github.com/rspivak/lbaws/blob/master/part2/wsgiapp.py>) directly and run the application under your Web server as:

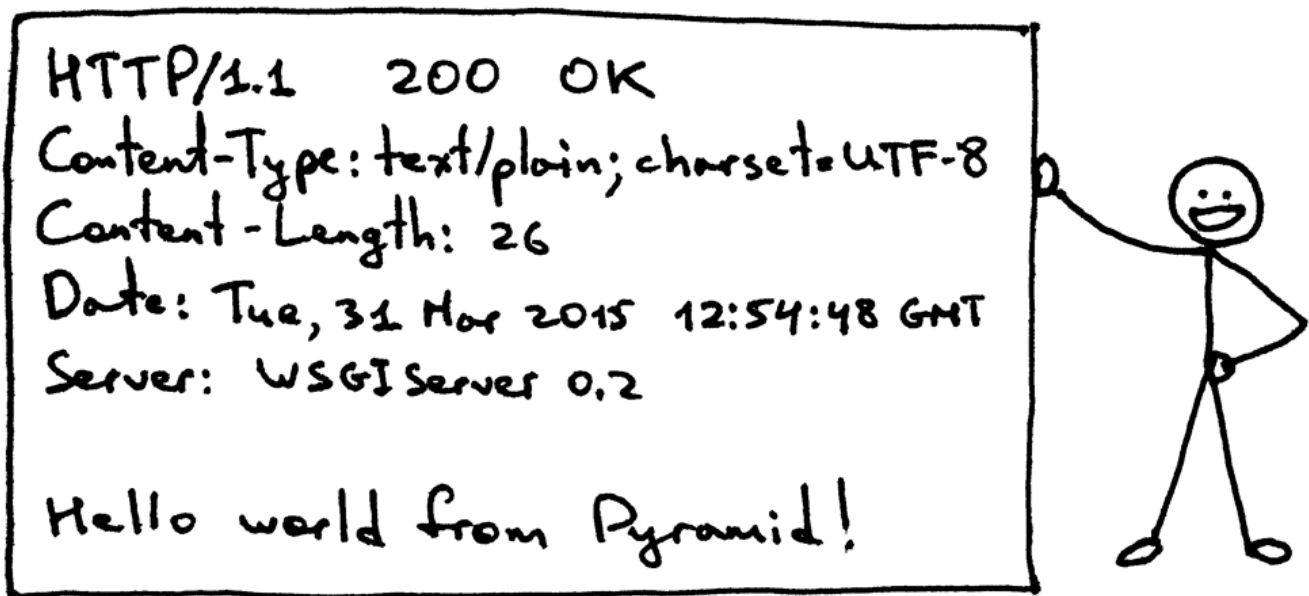
```
(lbaws) $ python webserver2.py wsgiapp:app
WSGIServer: Serving HTTP on port 8888 ...
```

Type in the following address and press Enter. This is the result you should see:



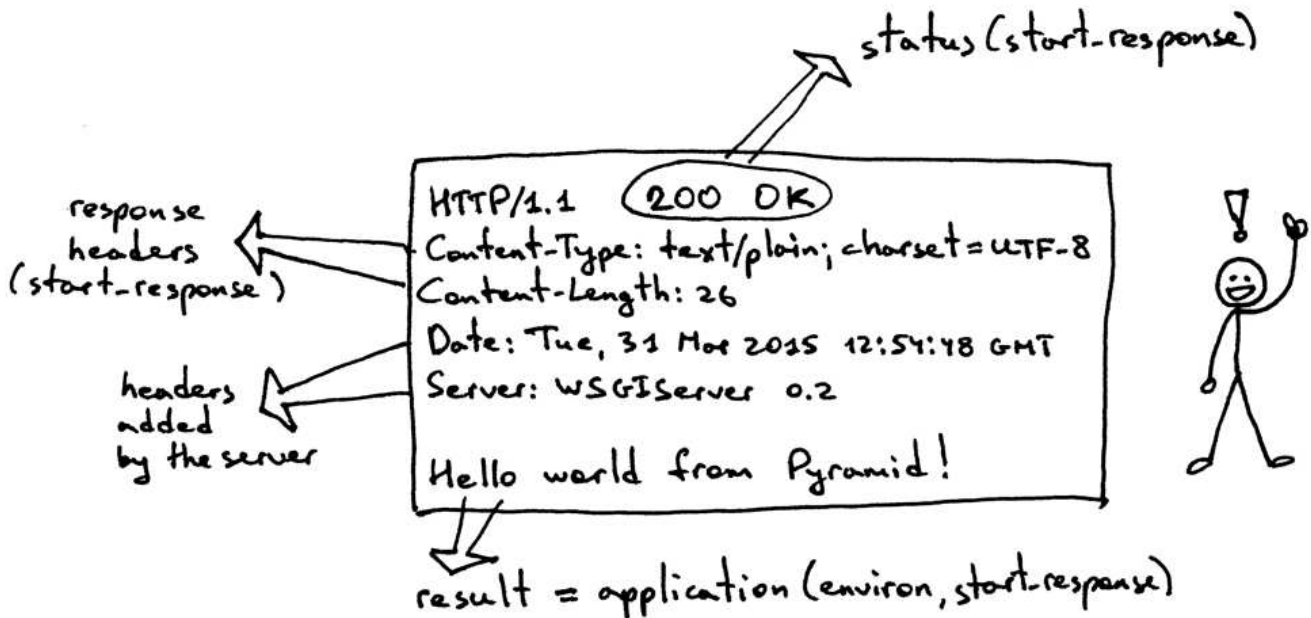
You just wrote your very own minimalistic WSGI Web framework while learning about how to create a Web server! Outrageous.

Now, let's get back to what the server transmits to the client. Here is the HTTP response the server generates when you call your Pyramid application using an HTTP client:



The response has some familiar parts that you saw in [Part 1](http://ruslanspivak.com/lbaws-part1/) (<http://ruslanspivak.com/lbaws-part1/>) but it also has something new. It has, for example, four [HTTP headers](http://en.wikipedia.org/wiki/List_of_HTTP_header_fields) (http://en.wikipedia.org/wiki/List_of_HTTP_header_fields) that you haven't seen before: *Content-Type*, *Content-Length*, *Date*, and *Server*. Those are the headers that a response from a Web server generally should have. None of them are strictly required, though. The purpose of the headers is to transmit additional information about the HTTP request/response.

Now that you know more about the WSGI interface, here is the same HTTP response with some more information about what parts produced it:



I haven't said anything about the 'environ' dictionary yet, but basically it's a Python dictionary that must contain certain WSGI and CGI variables prescribed by the WSGI specification. The server takes the values for the dictionary from the HTTP request after parsing the request. This is what the contents of the dictionary look like:

ENVIRON

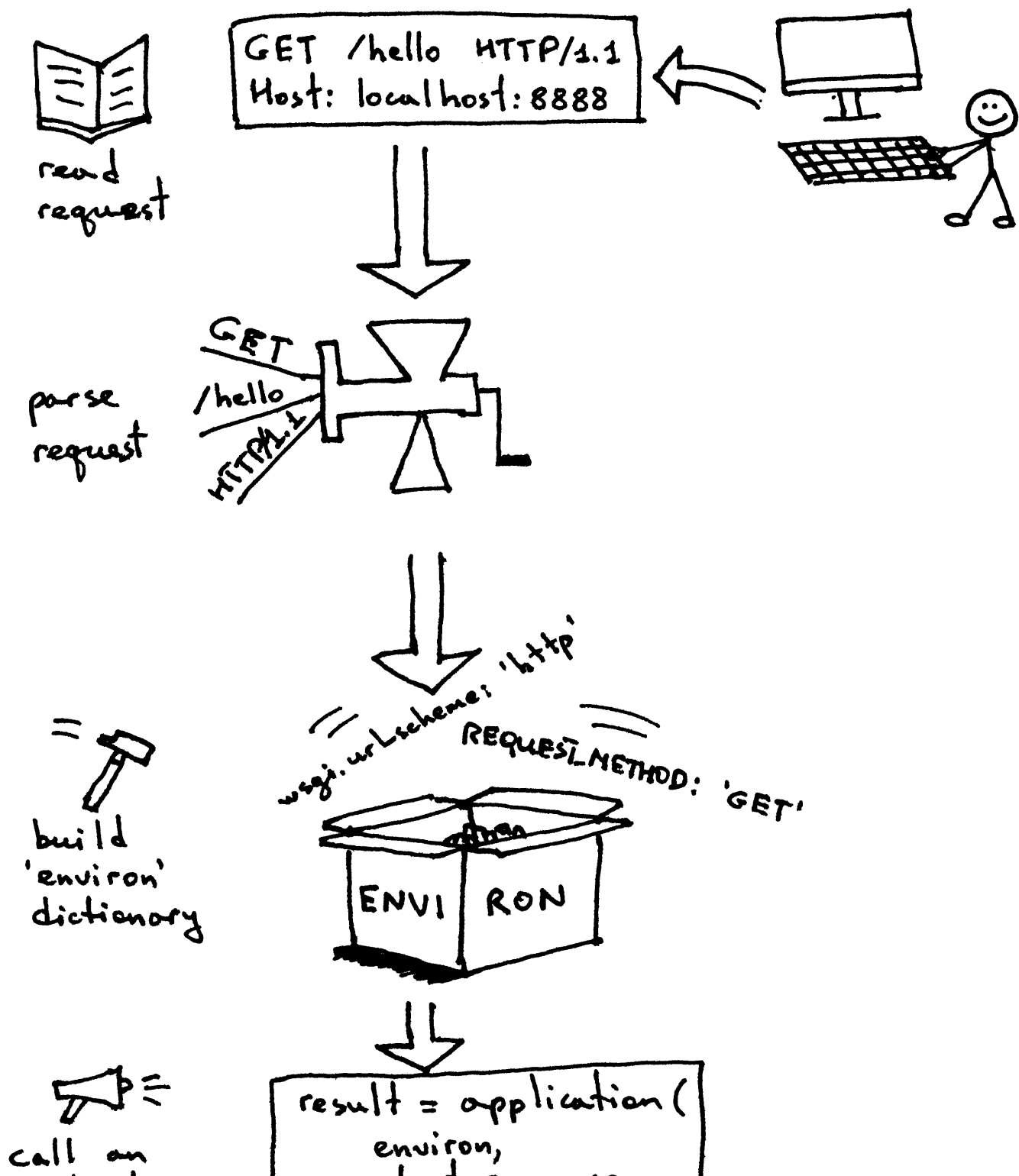
wsgi.version	(1, 0)
wsgi.url_scheme	'http'
wsgi.input	StringIO(request-data)
wsgi.errors	sys.stderr
wsgi.multithread	False
wsgi.multiprocess	False
wsgi.run-once	False
REQUEST_METHOD	'GET'
PATH_INFO	'/hello'
SERVER_NAME	'localhost'
SERVER_PORT	8888

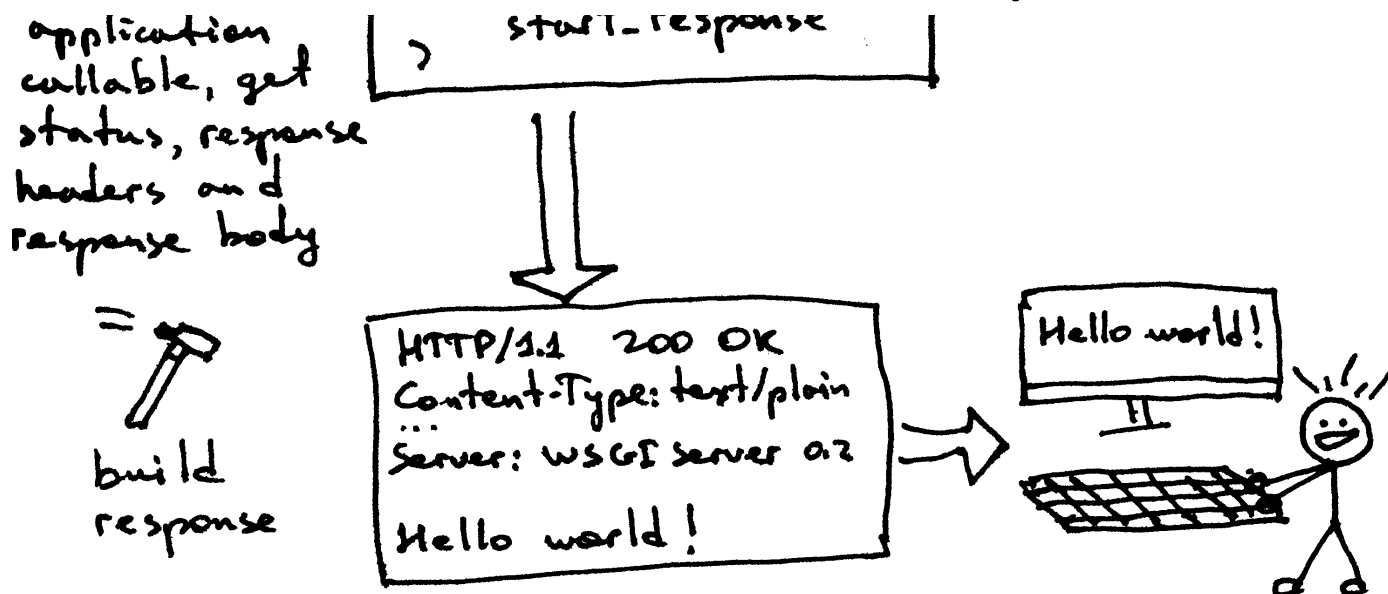
A Web framework uses the information from that dictionary to decide which view to use based on the specified route, request method etc., where to read the request body from and where to write errors, if any.

By now you've created your own WSGI Web server and you've made Web applications written with different Web frameworks. And, you've also created your barebones Web application/Web framework along the way. It's been a heck of a journey. Let's recap what your WSGI Web server has to do to

serve requests aimed at a WSGI application:

- First, the server starts and loads an *'application'* callable provided by your Web framework/application
- Then, the server reads a request
- Then, the server parses it
- Then, it builds an *'environ'* dictionary using the request data
- Then, it calls the *'application'* callable with the *'environ'* dictionary and a *'start_response'* callable as parameters and gets back a response body.
- Then, the server constructs an HTTP response using the data returned by the call to the *'application'* object and the status and response headers set by the *'start_response'* callable.
- And finally, the server transmits the HTTP response back to the client





That's about all there is to it. You now have a working WSGI server that can serve basic Web applications written with WSGI compliant Web frameworks like [Django](https://www.djangoproject.com/) (<https://www.djangoproject.com/>), [Flask](http://flask.pocoo.org/) (<http://flask.pocoo.org/>), [Pyramid](http://trypylramid.com/) (<http://trypylramid.com/>), or your very own WSGI framework. The best part is that the server can be used with multiple Web frameworks without any changes to the server code base. Not bad at all.

Before you go, here is another question for you to think about, "How do you make your server handle more than one request at a time?"

Stay tuned and I will show you a way to do that in Part 3. Cheers!

BTW, I'm writing a book "Let's Build A Web Server: First Steps" that explains how to write a basic web server from scratch and goes into more detail on topics I just covered. Subscribe to the mailing list to get the latest updates about the book and the release date.

Enter Your First Name *

Enter Your Best Email *

Get Updates!

All articles in this series:

- [Let's Build A Web Server. Part 1. \(http://ruslanspivak.com/lbaws-part1/\)](http://ruslanspivak.com/lbaws-part1/)
- [Let's Build A Web Server. Part 2. \(http://ruslanspivak.com/lbaws-part2/\)](http://ruslanspivak.com/lbaws-part2/)
- [Let's Build A Web Server. Part 3. \(http://ruslanspivak.com/lbaws-part3/\)](http://ruslanspivak.com/lbaws-part3/)

Comments

57 Comments Ruslan's Blog

Login

Recommend 19 Tweet Share

Sort by Best

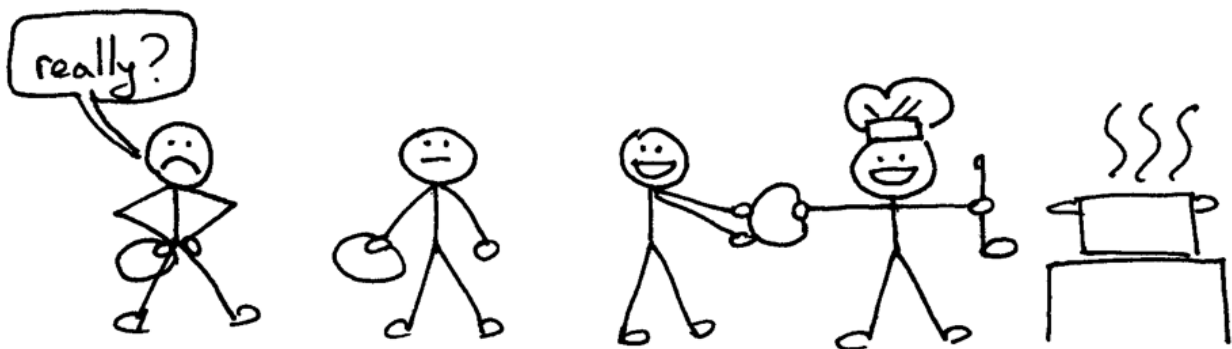
Let's Build A Web Server. Part 3. (<https://ruslanspivak.com/lbaws-part3/>)

Date 📅 Wed, May 20, 2015

"We learn most when we have to invent" —Piaget

In [Part 2 \(http://ruslanspivak.com/lbaws-part2/\)](http://ruslanspivak.com/lbaws-part2/) you created a minimalistic WSGI server that could handle basic HTTP GET requests. And I asked you a question, "How can you make your server handle more than one request at a time?" In this article you will find the answer. So, buckle up and shift into high gear. You're about to have a really fast ride. Have your Linux, Mac OS X (or any *nix system) and Python ready. All source code from the article is available on [GitHub \(https://github.com/rspivak/lbaws/blob/master/part3/\)](https://github.com/rspivak/lbaws/blob/master/part3/).

First let's remember what a very basic Web server looks like and what the server needs to do to service client requests. The server you created in [Part 1 \(http://ruslanspivak.com/lbaws-part1/\)](http://ruslanspivak.com/lbaws-part1/) and [Part 2 \(http://ruslanspivak.com/lbaws-part2/\)](http://ruslanspivak.com/lbaws-part2/) is an iterative server that handles one client request at a time. It cannot accept a new connection until after it has finished processing a current client request. Some clients might be unhappy with it because they will have to wait in line, and for busy servers the line might be too long.



Here is the code of the iterative server [webserver3a.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3a.py)
(<https://github.com/rspivak/lbaws/blob/master/part3/webserver3a.py>):

```
#####
# Iterative server - webserver3a.py                                     #
#                                                                       #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
#####
import socket

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5

def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

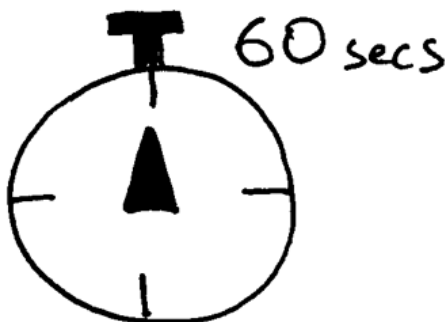
Hello, World!
"""
    client_connection.sendall(http_response)

def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    while True:
        client_connection, client_address = listen_socket.accept()
        handle_request(client_connection)
        client_connection.close()

if __name__ == '__main__':
    serve_forever()
```

To observe your server handling only one client request at a time, modify the server a little bit and add a 60 second delay after sending a response to a client. The change is only one line to tell the server process to sleep for 60 seconds.



And here is the code of the sleeping server `webserver3b.py`
<https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py>):

```
#####
# Iterative server - webserver3b.py                                     #
#                                                                     #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X   #
#                                                                     #
# - Server sleeps for 60 seconds after sending a response to a client #
#####
import socket
import time

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5

def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)
    time.sleep(60) # sleep and block the process for 60 seconds

def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    while True:
        client_connection, client_address = listen_socket.accept()
        handle_request(client_connection)
        client_connection.close()

if __name__ == '__main__':
    serve_forever()
```

Start the server with:

```
$ python webserver3b.py
```

Now open up a new terminal window and run the `curl` command. You should instantly see the “Hello, World!” string printed on the screen:

```
$ curl http://localhost:8888/hello
Hello, World!
```

And without delay open up a second terminal window and run the same `curl` command:

```
$ curl http://localhost:8888/hello
```

If you've done that within 60 seconds then the second *curl* should not produce any output right away and should just hang there. The server shouldn't print a new request body on its standard output either. Here is how it looks like on my Mac (the window at the bottom right corner highlighted in yellow shows the second *curl* command hanging, waiting for the connection to be accepted by the server):

A terminal window split into two panes. The left pane shows a Python web server running on port 8888, receiving a GET request for /hello. The right pane shows a curl command being executed, which prints 'Hello, World!' and then hangs. A second terminal window at the bottom right, highlighted in yellow, shows the same curl command hanging. The terminal status bar at the bottom indicates the session and the current command.

```
(lsbaws)Ruslans-MacBook-Air:part3 rspivak$ python webserver3b.py
Serving HTTP on port 8888 ...
GET /hello HTTP/1.1
User-Agent: curl/7.37.1
Host: localhost:8888
Accept: */*

Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello
Hello, World!

Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello

Session: 0 3 2 1: bash 2: bash- 3: curl*
"Ruslans-MacBook-Air.lo" 08:05 11-May-15
```

After you've waited long enough (more than 60 seconds) you should see the first *curl* terminate and the second *curl* print "*Hello, World!*" on the screen, then hang for 60 seconds, and then terminate:

```
(lsbaws)Ruslans-MacBook-Air:part3 rspivak$ python webserver3b.py
Serving HTTP on port 8888 ...
GET /hello HTTP/1.1
User-Agent: curl/7.37.1
Host: localhost:8888
Accept: */*

GET /hello HTTP/1.1
User-Agent: curl/7.37.1
Host: localhost:8888
Accept: */*

Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello
Hello, World!
Ruslans-MacBook-Air:~ rspivak$

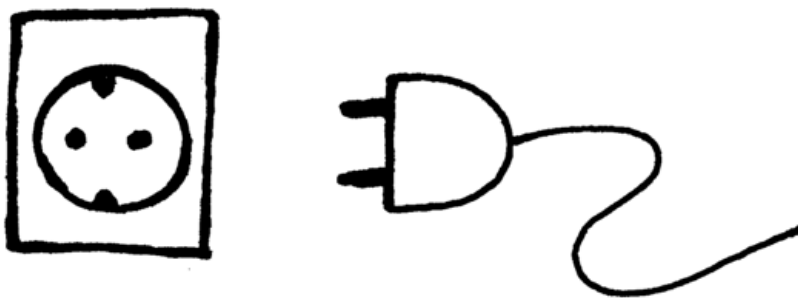
Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello
Hello, World!
Ruslans-MacBook-Air:~ rspivak$
```

Session: 0 3 2 1: bash 2: bash- 3: bash*

"Ruslans-MacBook-Air.lo" 08:20 11-May-15

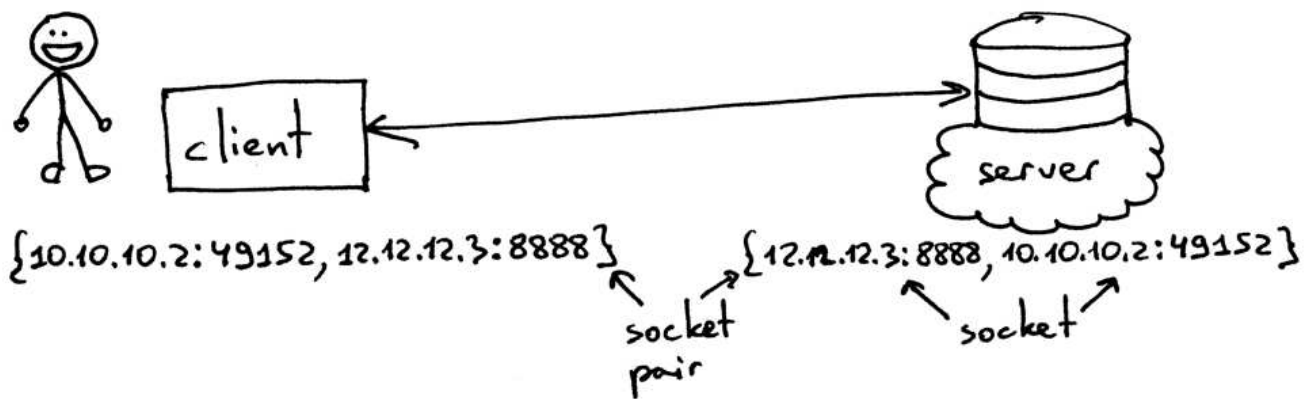
The way it works is that the server finishes servicing the first *curl* client request and then it starts handling the second request only after it sleeps for 60 seconds. It all happens sequentially, or iteratively, one step, or in our case one client request, at a time.

Let's talk about the communication between clients and servers for a bit. In order for two programs to communicate with each other over a network, they have to use sockets. And you saw sockets both in [Part 1 \(http://ruslanspivak.com/lsbaws-part1/\)](http://ruslanspivak.com/lsbaws-part1/) and [Part 2 \(http://ruslanspivak.com/lsbaws-part2/\)](http://ruslanspivak.com/lsbaws-part2/). But what is a socket?



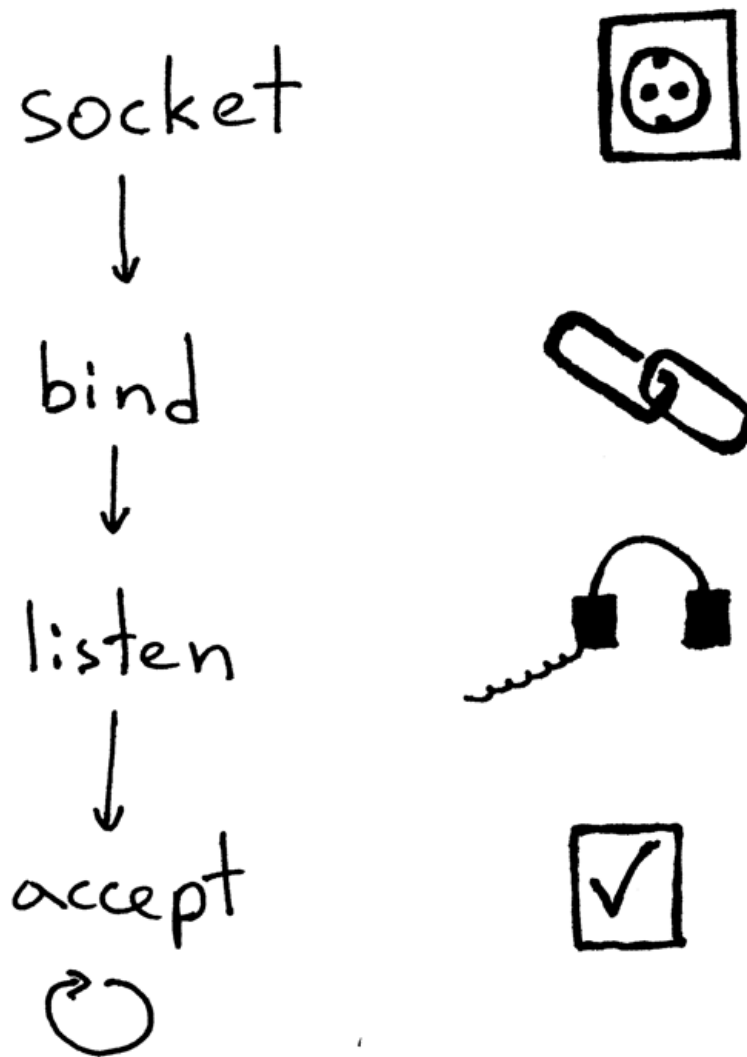
A *socket* is an abstraction of a communication endpoint and it allows your program to communicate with another program using file descriptors. In this article I'll be talking specifically about TCP/IP sockets on Linux/Mac OS X. An important notion to understand is the TCP socket pair.

The *socket pair* for a TCP connection is a 4-tuple that identifies two endpoints of the TCP connection: the local IP address, local port, foreign IP address, and foreign port. A socket pair uniquely identifies every TCP connection on a network. The two values that identify each endpoint, an IP address and a port number, are often called a *socket*.¹



So, the tuple $\{10.10.10.2:49152, 12.12.12.3:8888\}$ is a socket pair that uniquely identifies two endpoints of the TCP connection on the client and the tuple $\{12.12.12.3:8888, 10.10.10.2:49152\}$ is a socket pair that uniquely identifies the same two endpoints of the TCP connection on the server. The two values that identify the server endpoint of the TCP connection, the IP address 12.12.12.3 and the port 8888, are referred to as a socket in this case (the same applies to the client endpoint).

The standard sequence a server usually goes through to create a socket and start accepting client connections is the following:



1. The server creates a TCP/IP socket. This is done with the following statement in Python:

```
listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

2. The server might set some socket options (this is optional, but you can see that the server code above does just that to be able to re-use the same address over and over again if you decide to kill and re-start the server right away).

```
listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

3. Then, the server binds the address. The *bind* function assigns a local protocol address to the socket. With TCP, calling *bind* lets you specify a port number, an IP address, both, or neither.¹

```
listen_socket.bind(SERVER_ADDRESS)
```

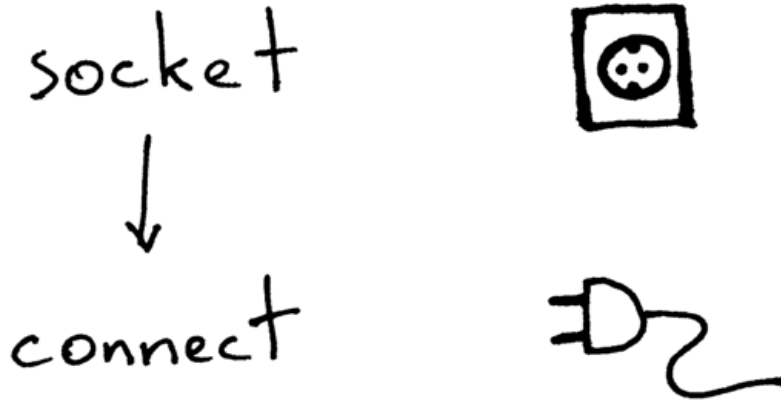
4. Then, the server makes the socket a listening socket

```
listen_socket.listen(REQUEST_QUEUE_SIZE)
```

The *listen* method is only called by *servers*. It tells the kernel that it should accept incoming connection requests for this socket.

After that's done, the server starts accepting client connections one connection at a time in a loop. When there is a connection available the *accept* call returns the connected client socket. Then, the server reads the request data from the connected client socket, prints the data on its standard output and sends a message back to the client. Then, the server closes the client connection and it is ready again to accept a new client connection.

Here is what a client needs to do to communicate with the server over TCP/IP:



Here is the sample code for a client to connect to your server, send a request and print the response:

```
import socket

# create a socket and connect to a server
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 8888))

# send and receive some data
sock.sendall(b'test')
data = sock.recv(1024)
print(data.decode())
```

After creating the socket, the client needs to connect to the server. This is done with the *connect* call:

```
sock.connect(('localhost', 8888))
```

The client only needs to provide the remote IP address or host name and the remote port number of a server to connect to.

You've probably noticed that the client doesn't call *bind* and *accept*. The client doesn't need to call *bind* because the client doesn't care about the local IP address and the local port number. The TCP/IP stack within the kernel automatically assigns the local IP address and the local port when the client calls *connect*. The local port is called an *ephemeral port*, i.e. a short-lived port.



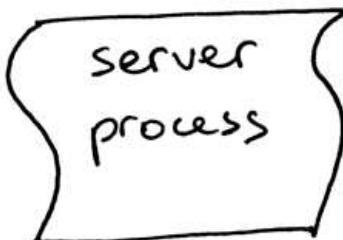
A port on a server that identifies a well-known service that a client connects to is called a *well-known* port (for example, 80 for HTTP and 22 for SSH). Fire up your Python shell and make a client connection to the server you run on localhost and see what ephemeral port the kernel assigns to the socket you've created (start the server [webserver3a.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3a.py) (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3a.py>) or [webserver3b.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py) (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py>) before trying the following example):

```
>>> import socket
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> sock.connect(('localhost', 8888))
>>> host, port = sock.getsockname()[:2]
>>> host, port
('127.0.0.1', 60589)
```

In the case above the kernel assigned the *ephemeral port* 60589 to the socket.

There are some other important concepts that I need to cover quickly before I get to answer the question from [Part 2](http://ruslanspivak.com/lbaws-part2/) (<http://ruslanspivak.com/lbaws-part2/>). You will see shortly why this is important. The two concepts are that of a *process* and a *file descriptor*.

What is a process? A *process* is just an instance of an executing program. When the server code is executed, for example, it's loaded into memory and an instance of that executing program is called a process. The kernel records a bunch of information about the process - its process ID would be one example - to keep track of it. When you run your iterative server [webserver3a.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3a.py) (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3a.py>) or [webserver3b.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py) (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py>) you run just one process.



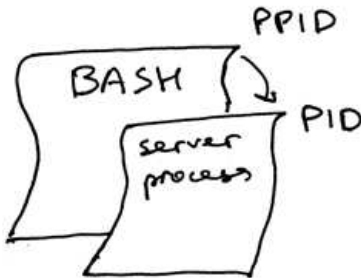
Start the server [webserver3b.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py) (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py>) in a terminal window:

```
$ python webserver3b.py
```

And in a different terminal window use the *ps* command to get the information about that process:

```
$ ps | grep webserver3b | grep -v grep
7182 ttys003    0:00.04 python webserver3b.py
```

The `ps` command shows you that you have indeed run just one Python process `webserver3b`. When a process gets created the kernel assigns a process ID to it, PID. In UNIX, every user process also has a parent that, in turn, has its own process ID called parent process ID, or PPID for short. I assume that you run a BASH shell by default and when you start the server, a new process gets created with a PID and its parent PID is set to the PID of the BASH shell.



Try it out and see for yourself how it all works. Fire up your Python shell again, which will create a new process, and then get the PID of the Python shell process and the parent PID (the PID of your BASH shell) using `os.getpid()` (<https://docs.python.org/2.7/library/os.html#os.getpid>) and `os.getppid()` (<https://docs.python.org/2.7/library/os.html#os.getppid>) system calls. Then, in another terminal window run `ps` command and grep for the PPID (parent process ID, which in my case is 3148). In the screenshot below you can see an example of a parent-child relationship between my child Python shell process and the parent BASH shell process on my Mac OS X:

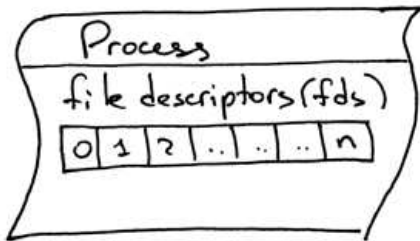
```
>>> import os
>>> os.getpid()
10236
>>> os.getppid()
3148
>>>
```

PID (points to 10236)
PPID (points to 3148)

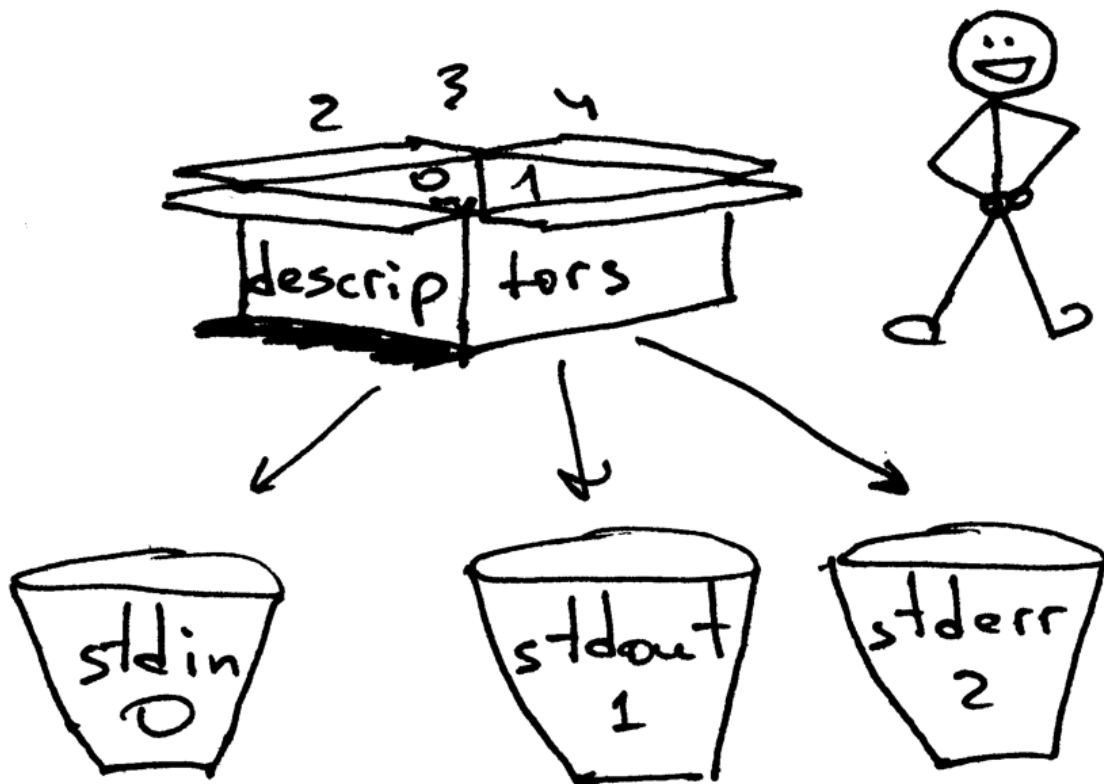
```
Ruslans-MacBook-Air:~ rspivak$ ps -opid,ppid,args | grep 3148 | grep -v grep
3148 1391 -bash
10236 3148 /usr/local/Cellar/python/2.7.9/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents/MacOS/Python
Ruslans-MacBook-Air:~ rspivak$
```

PID (points to 10236)
PPID (points to 3148)

Another important concept to know is that of a *file descriptor*. So what is a file descriptor? A *file descriptor* is a non-negative integer that the kernel returns to a process when it opens an existing file, creates a new file or when it creates a new socket. You've probably heard that in UNIX everything is a file. The kernel refers to the open files of a process by a file descriptor. When you need to read or write a file you identify it with the file descriptor. Python gives you high-level objects to deal with files (and sockets) and you don't have to use file descriptors directly to identify a file but, under the hood, that's how files and sockets are identified in UNIX: by their integer file descriptors.



By default, UNIX shells assign file descriptor 0 to the standard input of a process, file descriptor 1 to the standard output of the process and file descriptor 2 to the standard error.



As I mentioned before, even though Python gives you a high-level file or file-like object to work with, you can always use the `fileno()` method on the object to get the file descriptor associated with the file. Back to your Python shell to see how you can do that:

```
>>> import sys
>>> sys.stdin
<open file '<stdin>', mode 'r' at 0x102beb0c0>
>>> sys.stdin.fileno()
0
>>> sys.stdout.fileno()
1
>>> sys.stderr.fileno()
2
```

And while working with files and sockets in Python, you'll usually be using a high-level file/socket object, but there may be times where you need to use a file descriptor directly. Here is an example of how you can write a string to the standard output using a `write` (<https://docs.python.org/2.7/library/os.html#os.write>) system call that takes a file descriptor integer as a parameter:

```
>>> import sys
>>> import os
>>> res = os.write(sys.stdout.fileno(), 'hello\n')
hello
```

And here is an interesting part - which should not be surprising to you anymore because you already know that everything is a file in Unix - your socket also has a file descriptor associated with it. Again, when you create a socket in Python you get back an object and not a non-negative integer, but you can always get direct access to the integer file descriptor of the socket with the `fileno()` method that I mentioned earlier.

```
>>> import socket
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> sock.fileno()
3
```

One more thing I wanted to mention: have you noticed that in the second example of the iterative server `webserver3b.py` (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py>), when the server process was sleeping for 60 seconds you could still connect to the server with the second `curl` command? Sure, the `curl` didn't output anything right away and it was just hanging out there but how come the server was not *accept*ing a connection at the time and the client was not rejected right away, but instead was able to connect to the server? The answer to that is the `listen` method of a socket object and its `BACKLOG` argument, which I called `REQUEST_QUEUE_SIZE` in the code. The `BACKLOG` argument determines the size of a queue within the kernel for incoming connection requests. When the server `webserver3b.py` (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py>) was sleeping, the second `curl` command that you ran was able to connect to the server because the kernel had enough space available in the incoming connection request queue for the server socket.

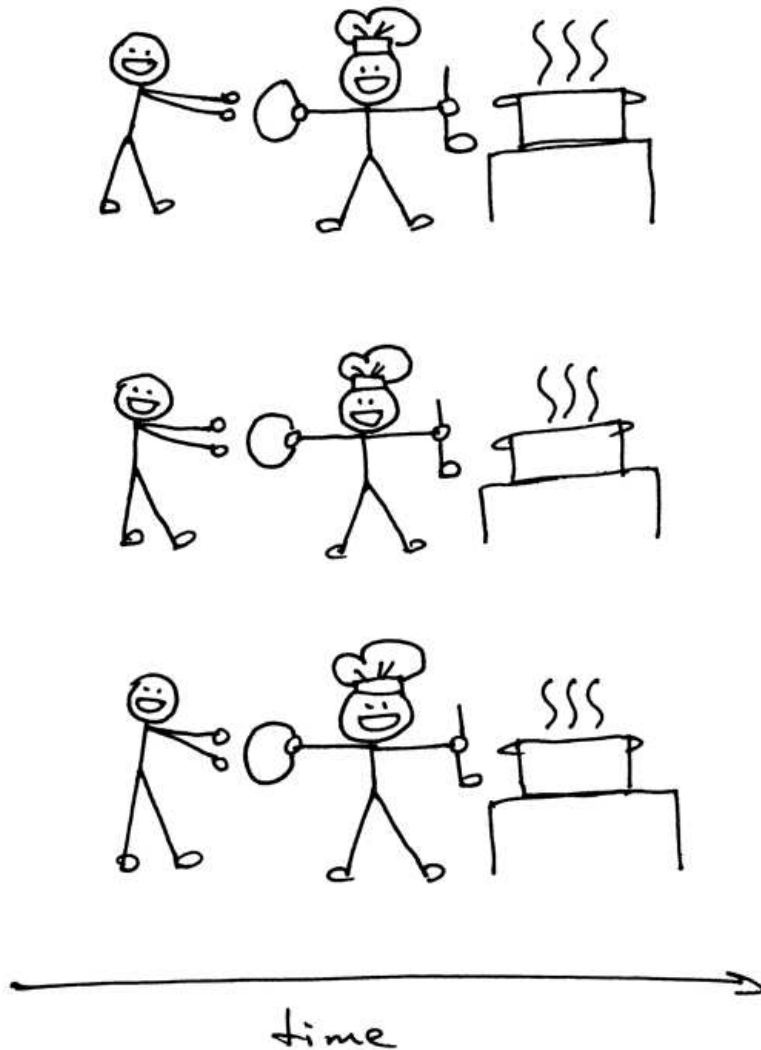
While increasing the `BACKLOG` argument does not magically turn your server into a server that can handle multiple client requests at a time, it is important to have a fairly large backlog parameter for busy servers so that the `accept` call would not have to wait for a new connection to be established but could grab the new connection off the queue right away and start processing a client request without delay.

Whoo-hoo! You've covered a lot of ground. Let's quickly recap what you've learned (or refreshed if it's all basics to you) so far.



- Iterative server
- Server socket creation sequence (socket, bind, listen, accept)
- Client connection creation sequence (socket, connect)
- Socket pair
- Socket
- Ephemeral port and well-known port
- Process
- Process ID (PID), parent process ID (PPID), and the parent-child relationship.
- File descriptors
- The meaning of the BACKLOG argument of the *listen* socket method

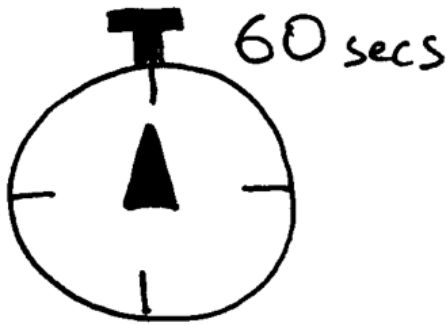
Now I am ready to answer the question from [Part 2 \(http://ruslanspivak.com/lbaws-part2/\)](http://ruslanspivak.com/lbaws-part2/): “How can you make your server handle more than one request at a time?” Or put another way, “How do you write a concurrent server?”



The simplest way to write a concurrent server under Unix is to use a `fork()` (<https://docs.python.org/2.7/library/os.html#os.fork>) system call.



Here is the code of your new shiny concurrent server [webserver3c.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3c.py) (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3c.py>) that can handle multiple client requests at the same time (as in our iterative server example [webserver3b.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py) (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py>), every child process sleeps for 60 secs):



```
#####
# Concurrent server - webserver3c.py #
# #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
# #
# - Child process sleeps for 60 seconds after handling a client's request #
# - Parent and child processes close duplicate descriptors #
# #
#####
import os
import socket
import time

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5

def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(
        'Child PID: {pid}. Parent PID {ppid}'.format(
            pid=os.getpid(),
            ppid=os.getppid(),
        )
    )
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)
    time.sleep(60)

def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))
    print('Parent PID (PPID): {pid}\n'.format(pid=os.getpid()))

    while True:
        client_connection, client_address = listen_socket.accept()
        pid = os.fork()
        if pid == 0: # child
            listen_socket.close() # close child copy
            handle_request(client_connection)
            client_connection.close()
            os._exit(0) # child exits here
        else: # parent
            client_connection.close() # close parent copy and loop over

if __name__ == '__main__':
    serve_forever()
```

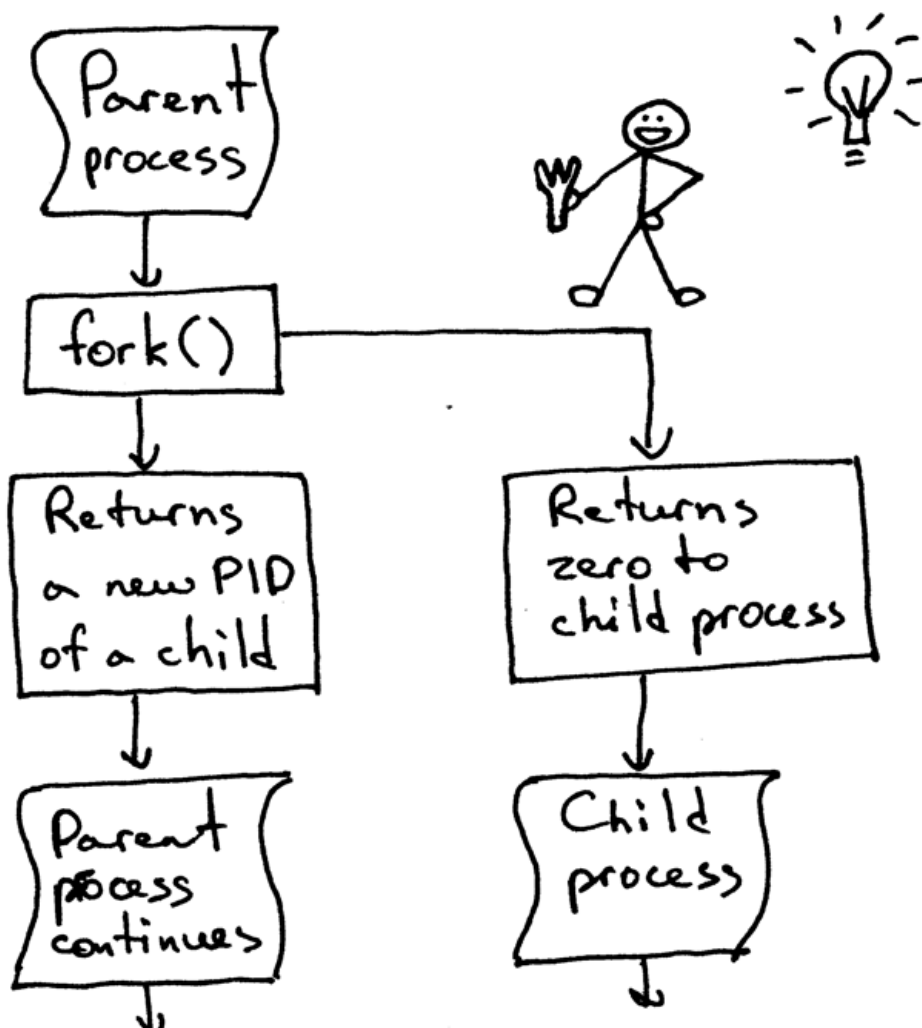
Before diving in and discussing how *fork* works, try it, and see for yourself that the server can indeed handle multiple client requests at the same time, unlike its iterative counterparts `webserver3a.py` (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3a.py>) and `webserver3b.py` (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3b.py>). Start the server on the command line with:

```
$ python webserver3c.py
```

And try the same two *curl* commands you've tried before with the iterative server and see for yourself that, now, even though the server child process sleeps for 60 seconds after serving a client request, it doesn't affect other clients because they are served by different and completely independent processes. You should see your *curl* commands output "Hello, World!" instantly and then hang for 60 secs. You can keep on running as many *curl* commands as you want (well, almost as many as you want :) and all of them will output the server's response "Hello, World" immediately and without any noticeable delay. Try it.

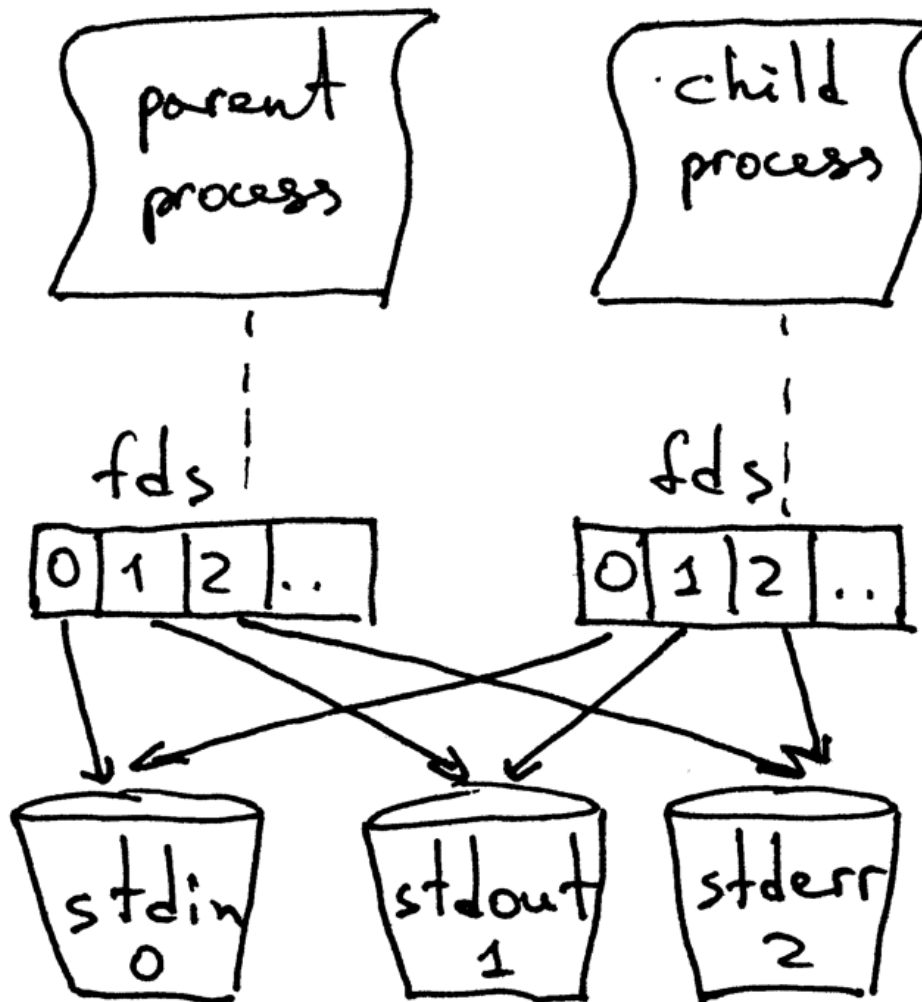
The most important point to understand about `fork()`

(<https://docs.python.org/2.7/library/os.html#os.fork>) is that you call *fork* once but it returns twice: once in the parent process and once in the child process. When you fork a new process the process ID returned to the child process is 0. When the *fork* returns in the parent process it returns the child's PID.



I still remember how fascinated I was by *fork* when I first read about it and tried it. It looked like magic to me. Here I was reading a sequential code and then “boom!”: the code cloned itself and now there were two instances of the same code running concurrently. I thought it was nothing short of magic, seriously.

When a parent forks a new child, the child process gets a copy of the parent's file descriptors:



You've probably noticed that the parent process in the code above closed the client connection:

```
else: # parent
    client_connection.close() # close parent copy and loop over
```

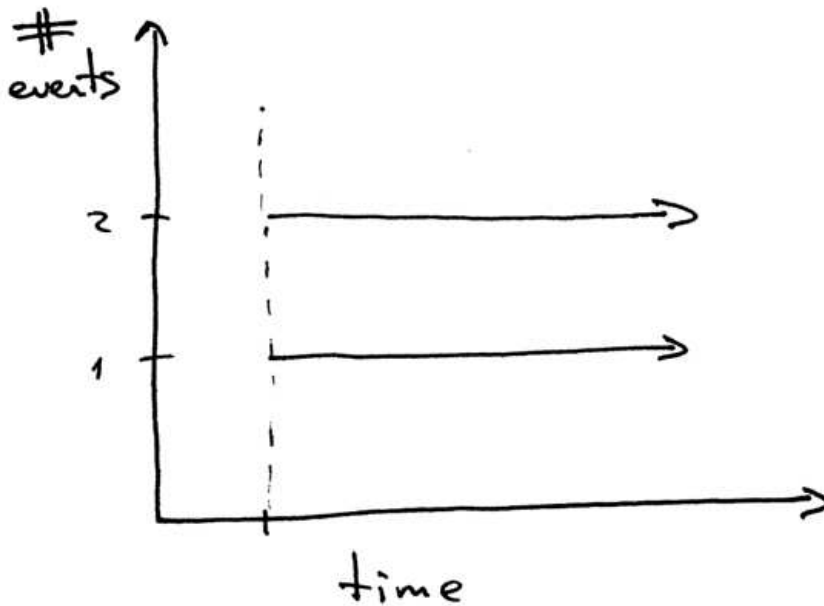
So how come a child process is still able to read the data from a client socket if its parent closed the very same socket? The answer is in the picture above. The kernel uses descriptor reference counts to decide whether to close a socket or not. It closes the socket only when its descriptor reference count becomes 0. When your server creates a child process, the child gets the copy of the parent's file descriptors and the kernel increments the reference counts for those descriptors. In the case of one parent and one child, the descriptor reference count would be 2 for the client socket and when the parent process in the code above closes the client connection socket, it merely decrements its reference count which becomes 1, not small enough to cause the kernel to close the socket. The child process also closes the duplicate copy of the parent's *listen_socket* because the child doesn't care about accepting new client connections, it cares only about processing requests from the established client connection:

```
listen_socket.close() # close child copy
```

I'll talk about what happens if you do not close duplicate descriptors later in the article.

As you can see from the source code of your concurrent server, the sole role of the server parent process now is to accept a new client connection, fork a new child process to handle that client request, and loop over to accept another client connection, and nothing more. The server parent process does not process client requests - its children do.

A little aside. What does it mean when we say that two events are concurrent?



When we say that two events are concurrent we usually mean that they happen at the same time. As a shorthand that definition is fine, but you should remember the strict definition:

Two events are *concurrent* if you cannot tell by looking at the program which will happen first.²

Again, it's time to recap the main ideas and concepts you've covered so far.



- The simplest way to write a concurrent server in Unix is to use the `fork()` (<https://docs.python.org/2.7/library/os.html#os.fork>) system call
- When a process forks a new process it becomes a parent process to that newly forked child process.
- Parent and child share the same file descriptors after the call to `fork`.
- The kernel uses descriptor reference counts to decide whether to close the file/socket or not
- The role of a server parent process: all it does now is accept a new connection from a client, fork a child to handle the client request, and loop over to accept a new client connection.

Let's see what is going to happen if you don't close duplicate socket descriptors in the parent and child processes. Here is a modified version of the concurrent server where the server does not close duplicate descriptors, [webserver3d.py](#)

(<https://github.com/rspivak/lbaws/blob/master/part3/webserver3d.py>):

```
#####
# Concurrent server - webserver3d.py                                     #
#                                                                       #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X    #
#####
import os
import socket

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5

def handle_request(client_connection):
    request = client_connection.recv(1024)
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)

def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    clients = []
    while True:
        client_connection, client_address = listen_socket.accept()
        # store the reference otherwise it's garbage collected
        # on the next loop run
        clients.append(client_connection)
        pid = os.fork()
        if pid == 0: # child
            listen_socket.close() # close child copy
            handle_request(client_connection)
            client_connection.close()
            os._exit(0) # child exits here
        else: # parent
            # client_connection.close()
            print(len(clients))

if __name__ == '__main__':
    serve_forever()
```

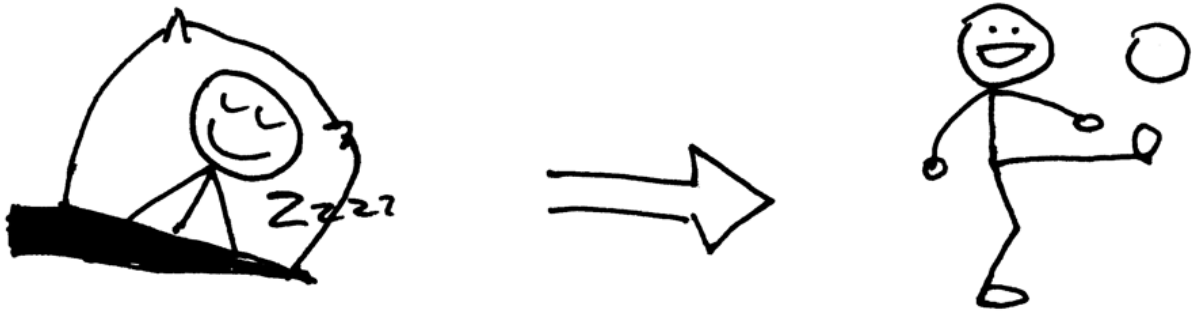
Start the server with:

```
$ python webserver3d.py
```

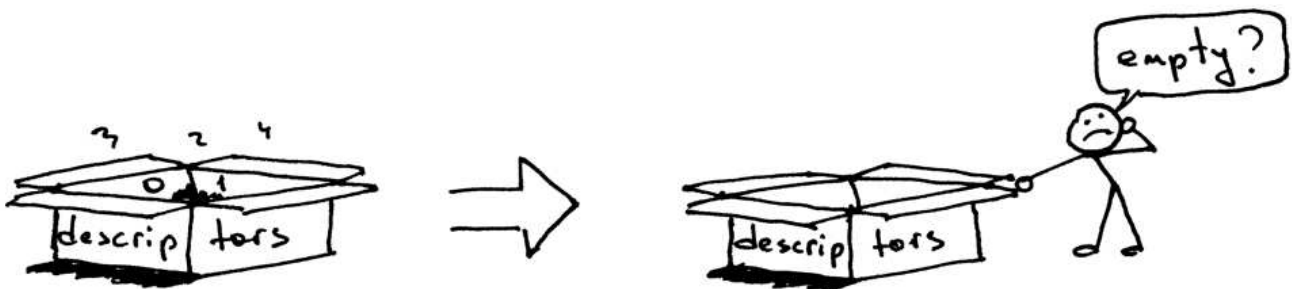
Use *curl* to connect to the server:

```
$ curl http://localhost:8888/hello
Hello, World!
```

Okay, the *curl* printed the response from the concurrent server but it did not terminate and kept hanging. What is happening here? The server no longer sleeps for 60 seconds: its child process actively handles a client request, closes the client connection and exits, but the client *curl* still does not terminate.



So why does the *curl* not terminate? The reason is the duplicate file descriptors. When the child process closed the client connection, the kernel decremented the reference count of that client socket and the count became 1. The server child process exited, but the client socket was not closed by the kernel because the reference count for that socket descriptor was not 0, and, as a result, the termination packet (called FIN in TCP/IP parlance) was not sent to the client and the client stayed on the line, so to speak. There is also another problem. If your long-running server doesn't close duplicate file descriptors, it will eventually run out of available file descriptors:



Stop your server [webserver3d.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3d.py)

(<https://github.com/rspivak/lbaws/blob/master/part3/webserver3d.py>) with *Control-C* and check out the default resources available to your server process set up by your shell with the shell built-in command *ulimit*:

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 3842
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files              (-n) 1024
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 3842
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
```

As you can see above, the maximum number of open file descriptors (*open files*) available to the server process on my Ubuntu box is 1024.

Now let's see how your server can run out of available file descriptors if it doesn't close duplicate descriptors. In an existing or new terminal window, set the maximum number of open file descriptors for your server to be 256:

```
$ ulimit -n 256
```

Start the server `webserver3d.py`

(<https://github.com/rspivak/lbaws/blob/master/part3/webserver3d.py>) in the same terminal where you've just run the `$ ulimit -n 256` command:

```
$ python webserver3d.py
```

and use the following client `client3.py` (<https://github.com/rspivak/lbaws/blob/master/part3/client3.py>) to test the server.

```
#####
# Test client - client3.py                                     #
#                                                             #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
#####
import argparse
import errno
import os
import socket

SERVER_ADDRESS = 'localhost', 8888
REQUEST = b"""\
GET /hello HTTP/1.1
Host: localhost:8888

"""

def main(max_clients, max_conns):
    socks = []
    for client_num in range(max_clients):
        pid = os.fork()
        if pid == 0:
            for connection_num in range(max_conns):
                sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                sock.connect(SERVER_ADDRESS)
                sock.sendall(REQUEST)
                socks.append(sock)
                print(connection_num)
            os._exit(0)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Test client for LSBAWS.',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter,
    )
    parser.add_argument(
        '--max-conns',
        type=int,
        default=1024,
        help='Maximum number of connections per client.'
    )
    parser.add_argument(
        '--max-clients',
        type=int,
        default=1,
        help='Maximum number of clients.'
    )
    args = parser.parse_args()
    main(args.max_clients, args.max_conns)
```

In a new terminal window, start the `client3.py` (<https://github.com/rspivak/lbaws/blob/master/part3/client3.py>) and tell it to create 300 simultaneous connections to the server:

```
$ python client3.py --max-clients=300
```

Soon enough your server will explode. Here is a screenshot of the exception on my box:

```
248
249
250
251
252
Traceback (most recent call last):
  File "webserver3d.py", line 58, in <module>
  File "webserver3d.py", line 43, in serve_forever
  File "/usr/lib/python2.7/socket.py", line 202, in accept
socket.error: [Errno 24] Too many open files
```

The lesson is clear - your server should close duplicate descriptors. But even if you close duplicate descriptors, you are not out of the woods yet because there is another problem with your server, and that problem is zombies!



Yes, your server code actually creates zombies. Let's see how. Start up your server again:

```
$ python webserver3d.py
```

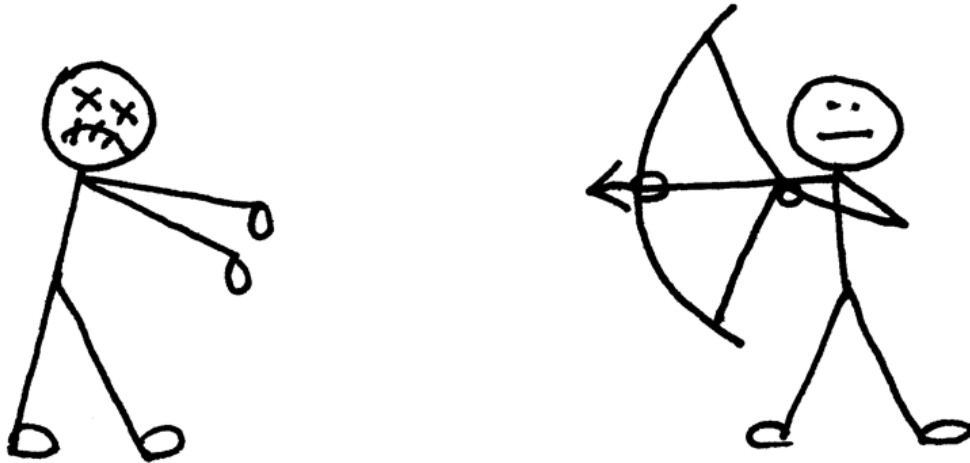
Run the following `curl` command in another terminal window:

```
$ curl http://localhost:8888/hello
```

And now run the `ps` command to show running Python processes. This the example of `ps` output on my Ubuntu box:

```
$ ps auxw | grep -i python | grep -v grep
vagrant  9099  0.0  1.2 31804 6256 pts/0    S+   16:33   0:00 python webserver3d.py
vagrant  9102  0.0  0.0      0      0 pts/0    Z+   16:33   0:00 [python] <defunct>
```

Do you see the second line above where it says the status of the process with PID 9102 is **Z+** and the name of the process is **<defunct>**? That's our zombie there. The problem with zombies is that you can't kill them.



Even if you try to kill zombies with `$ kill -9`, they will survive. Try it and see for yourself.

What is a zombie anyway and why does our server create them? A *zombie* is a process that has terminated, but its parent has not *waited* for it and has not received its termination status yet. When a child process exits before its parent, the kernel turns the child process into a zombie and stores some information about the process for its parent process to retrieve later. The information stored is usually the process ID, the process termination status, and the resource usage by the process. Okay, so zombies serve a purpose, but if your server doesn't take care of these zombies your system will get clogged up. Let's see how that happens. First stop your running server and, in a new terminal window, use the `ulimit` command to set the *max user processes* to 400 (make sure to set *open files* to a high number, let's say 500 too):

```
$ ulimit -u 400
$ ulimit -n 500
```

Start the server `webserver3d.py`

(<https://github.com/rspivak/lbaws/blob/master/part3/webserver3d.py>) in the same terminal where you've just run the `$ ulimit -u 400` command:

```
$ python webserver3d.py
```

In a new terminal window, start the `client3.py`

(<https://github.com/rspivak/lbaws/blob/master/part3/client3.py>) and tell it to create 500 simultaneous connections to the server:

```
$ python client3.py --max-clients=500
```

And, again, soon enough your server will blow up with an **OSError: Resource temporarily unavailable** exception when it tries to create a new child process, but it can't because it has reached the limit for the maximum number of child processes it's allowed to create. Here is a screenshot of the exception on my box:

```
186
187
188
189
190
191
Traceback (most recent call last):
  File "webserver3d.py", line 58, in <module>
    serve_forever()
  File "webserver3d.py", line 47, in serve_forever
    pid = os.fork()
OSError: [Errno 11] Resource temporarily unavailable
```

As you can see, zombies create problems for your long-running server if it doesn't take care of them. I will discuss shortly how the server should deal with that zombie problem.

Let's recap the main points you've covered so far:

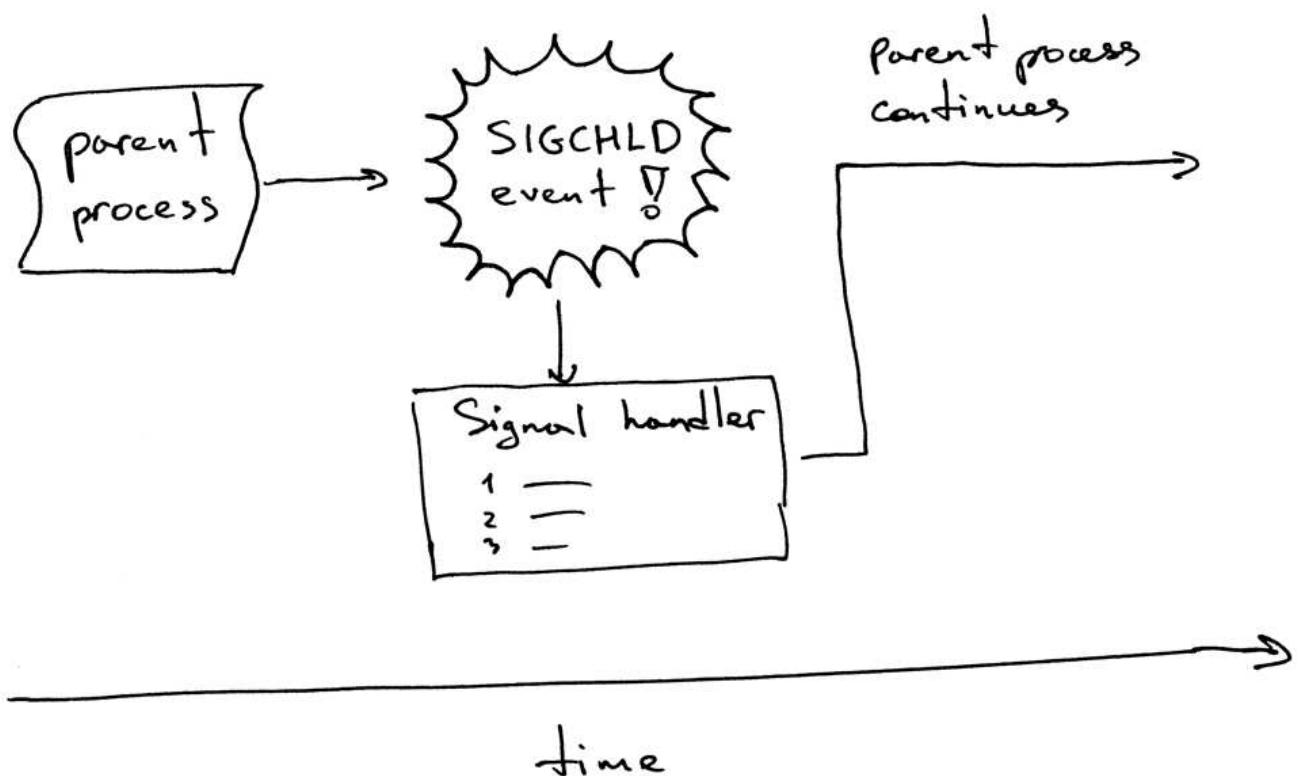


- If you don't close duplicate descriptors, the clients won't terminate because the client connections won't get closed.
- If you don't close duplicate descriptors, your long-running server will eventually run out of available file descriptors (*max open files*).
- When you fork a child process and it exits and the parent process doesn't *wait* for it and doesn't collect its termination status, it becomes a *zombie*.
- Zombies need to eat something and, in our case, it's memory. Your server will eventually run out of available processes (*max user processes*) if it doesn't take care of zombies.
- You can't *kill* a zombie, you need to *wait* for it.

So what do you need to do to take care of zombies? You need to modify your server code to *wait* for zombies to get their termination status. You can do that by modifying your server to call a [wait](https://docs.python.org/2.7/library/os.html#os.wait) (<https://docs.python.org/2.7/library/os.html#os.wait>) system call. Unfortunately, that's far from ideal because if you call *wait* and there is no terminated child process the call to *wait* will block your server, effectively preventing your server from handling new client connection requests. Are there any other options? Yes, there are, and one of them is the combination of a *signal handler* with the *wait* system call.



Here is how it works. When a child process exits, the kernel sends a *SIGCHLD* signal. The parent process can set up a signal handler to be asynchronously notified of that *SIGCHLD* event and then it can *wait* for the child to collect its termination status, thus preventing the zombie process from being left around.



By the way, an asynchronous event means that the parent process doesn't know ahead of time that the event is going to happen.

Modify your server code to set up a *SIGCHLD* event handler and *wait* for a terminated child in the event handler. The code is available in [webserver3e.py](https://github.com/rspivak/lbaws/blob/master/part3/webserver3e.py)

(<https://github.com/rspivak/lbaws/blob/master/part3/webserver3e.py>) file:


```
#####
# Concurrent server - webserver3e.py                                     #
#                                                                       #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X    #
#####
import os
import signal
import socket
import time

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 5

def grim_reaper(signum, frame):
    pid, status = os.wait()
    print(
        'Child {pid} terminated with status {status}'
        '\n'.format(pid=pid, status=status)
    )

def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)
    # sleep to allow the parent to loop over to 'accept' and block there
    time.sleep(3)

def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    signal.signal(signal.SIGCHLD, grim_reaper)

    while True:
        client_connection, client_address = listen_socket.accept()
        pid = os.fork()
        if pid == 0: # child
            listen_socket.close() # close child copy
            handle_request(client_connection)
            client_connection.close()
            os._exit(0)
        else: # parent
            client_connection.close()
```

```
if __name__ == '__main__':  
    serve_forever()
```

Start the server:

```
$ python webserver3e.py
```

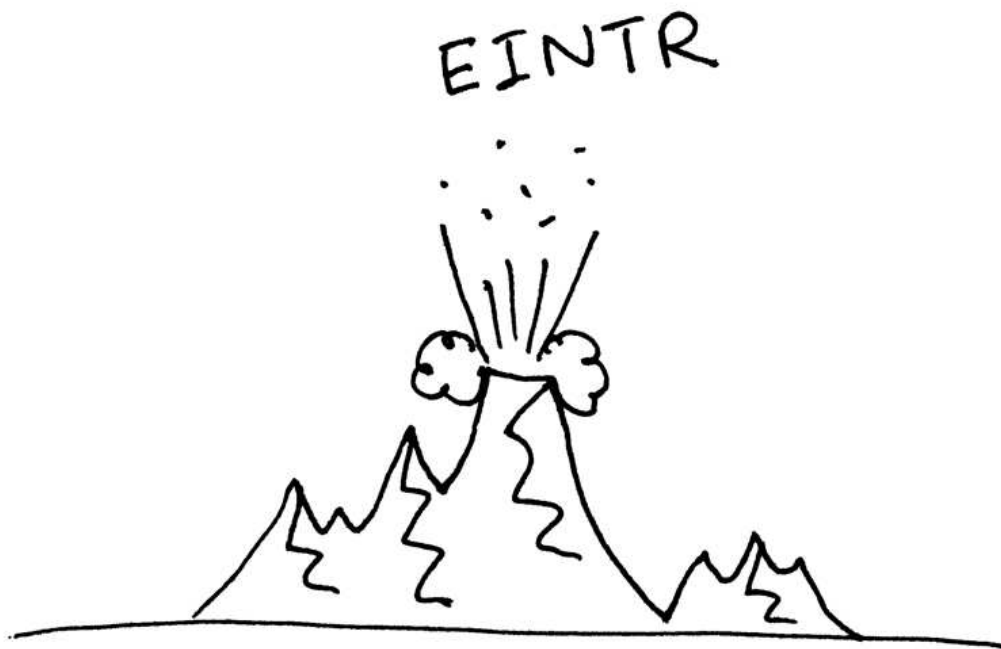
Use your old friend *curl* to send a request to the modified concurrent server:

```
$ curl http://localhost:8888/hello
```

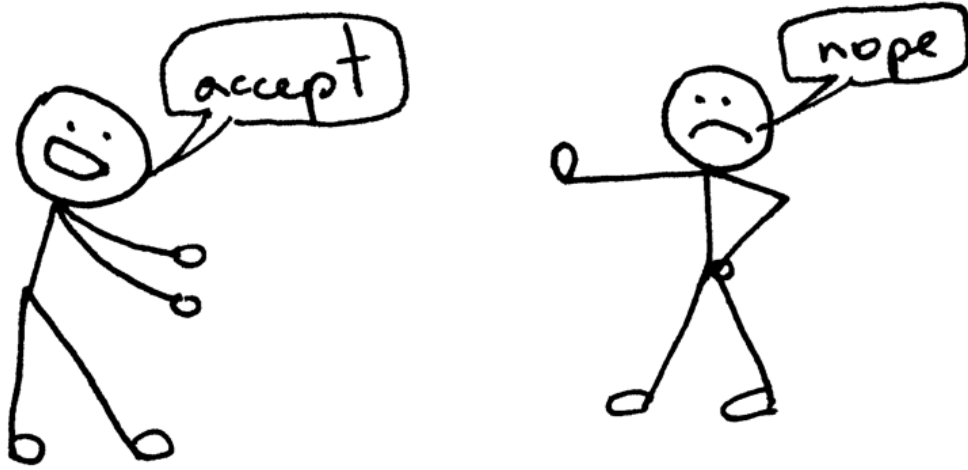
Look at the server:

```
Serving HTTP on port 8888 ...  
GET /hello HTTP/1.1  
User-Agent: curl/7.35.0  
Host: localhost:8888  
Accept: */*  
  
Child 9951 terminated with status 0  
  
Traceback (most recent call last):  
  File "webserver3e.py", line 62, in <module>  
    serve_forever()  
  File "webserver3e.py", line 51, in serve_forever  
    client_connection, client_address = listen_socket.accept()  
  File "/usr/lib/python2.7/socket.py", line 202, in accept  
    sock, addr = self._sock.accept()  
socket.error: [Errno 4] Interrupted system call
```

What just happened? The call to *accept* failed with the error *EINTR*.



The parent process was blocked in *accept* call when the child process exited which caused *SIGCHLD* event, which in turn activated the signal handler and when the signal handler finished the *accept* system call got interrupted:



Don't worry, it's a pretty simple problem to solve, though. All you need to do is to re-start the *accept* system call. Here is the modified version of the server `webserver3f.py` (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3f.py>) that handles that problem:

```
#####
# Concurrent server - webserver3f.py                                     #
#                                                                       #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X    #
#####
import errno
import os
import signal
import socket

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 1024

def grim_reaper(signum, frame):
    pid, status = os.wait()

def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)

def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    signal.signal(signal.SIGCHLD, grim_reaper)

    while True:
        try:
            client_connection, client_address = listen_socket.accept()
        except IOError as e:
            code, msg = e.args
            # restart 'accept' if it was interrupted
            if code == errno.EINTR:
                continue
            else:
                raise

        pid = os.fork()
        if pid == 0: # child
            listen_socket.close() # close child copy
            handle_request(client_connection)
            client_connection.close()
            os._exit(0)
        else: # parent
```

```
client_connection.close() # close parent copy and loop over
```

```
if __name__ == '__main__':  
    serve_forever()
```

Start the updated server `webserver3f.py`

(<https://github.com/rspivak/lbaws/blob/master/part3/webserver3f.py>):

```
$ python webserver3f.py
```

Use `curl` to send a request to the modified concurrent server:

```
$ curl http://localhost:8888/hello
```

See? No `EINTR` exceptions any more. Now, verify that there are no more zombies either and that your `SIGCHLD` event handler with `wait` call took care of terminated children. To do that, just run the `ps` command and see for yourself that there are no more Python processes with **Z+** status (no more **<defunct>** processes). Great! It feels safe without zombies running around.



- If you *fork* a child and don't wait for it, it becomes a *zombie*.
- Use the `SIGCHLD` event handler to asynchronously *wait* for a terminated child to get its termination status
- When using an event handler you need to keep in mind that system calls might get interrupted and you need to be prepared for that scenario

Okay, so far so good. No problems, right? Well, almost. Try your `webserver3f.py` (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3f.py>) again, but instead of making one request with `curl` use `client3.py` (<https://github.com/rspivak/lbaws/blob/master/part3/client3.py>) to create 128 simultaneous connections:

```
$ python client3.py --max-clients 128
```

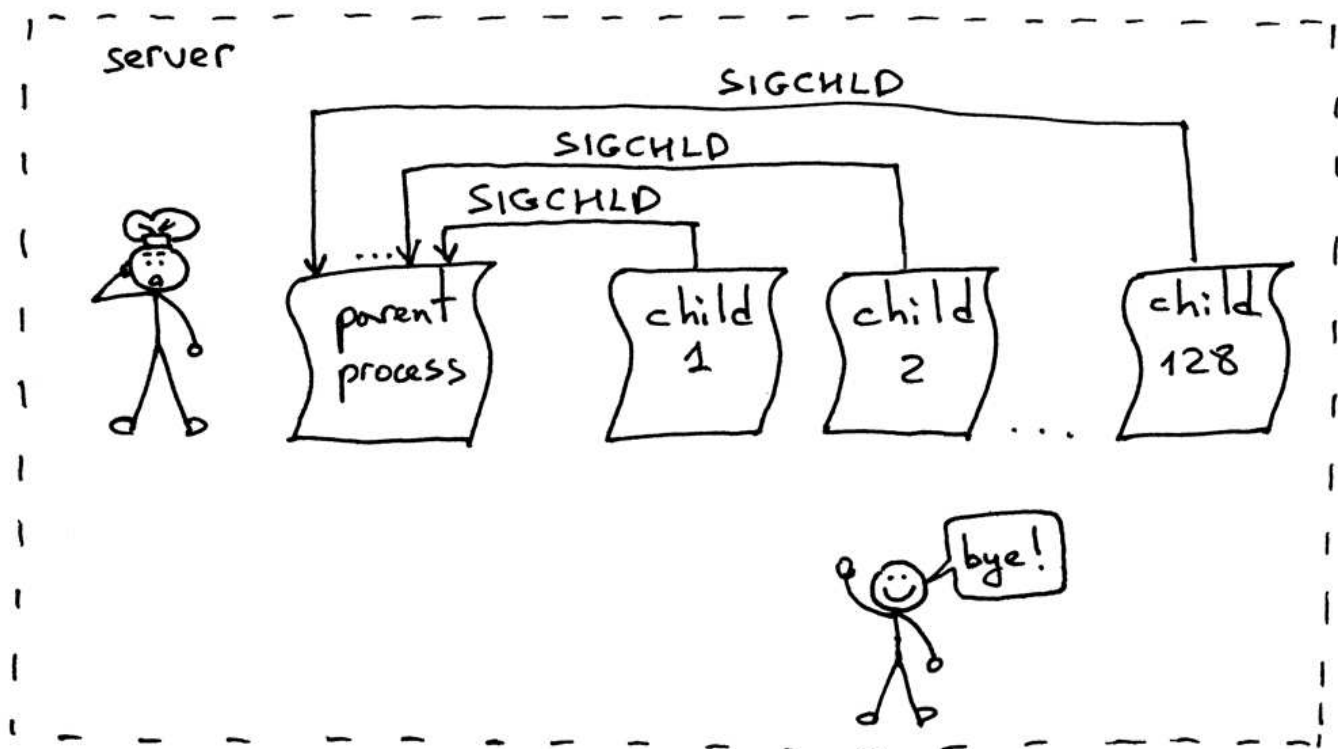
Now run the `ps` command again

```
$ ps auxw | grep -i python | grep -v grep
```

and see that, oh boy, zombies are back again!



What went wrong this time? When you ran 128 simultaneous clients and established 128 connections, the child processes on the server handled the requests and exited almost at the same time causing a flood of `SIGCHLD` signals being sent to the parent process. The problem is that the signals are not queued and your server process missed several signals, which left several zombies running around unattended:



The solution to the problem is to set up a `SIGCHLD` event handler but instead of `wait` use a `waitpid` (<https://docs.python.org/2.7/library/os.html#os.waitpid>) system call with a `WNOHANG` option in a loop to make sure that all terminated child processes are taken care of. Here is the modified server code, `webserver3g.py` (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3g.py>):

```
#####
# Concurrent server - webserver3g.py                                     #
#                                                                       #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X    #
#####
import errno
import os
import signal
import socket

SERVER_ADDRESS = (HOST, PORT) = '', 8888
REQUEST_QUEUE_SIZE = 1024

def grim_reaper(signum, frame):
    while True:
        try:
            pid, status = os.waitpid(
                -1,          # Wait for any child process
                os.WNOHANG   # Do not block and return EWOULDBLOCK error
            )
        except OSError:
            return

        if pid == 0: # no more zombies
            return

def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""\
HTTP/1.1 200 OK

Hello, World!
"""
    client_connection.sendall(http_response)

def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {port} ...'.format(port=PORT))

    signal.signal(signal.SIGCHLD, grim_reaper)

    while True:
        try:
            client_connection, client_address = listen_socket.accept()
        except IOError as e:
            code, msg = e.args
            # restart 'accept' if it was interrupted
            if code == errno.EINTR:
                continue
```

```
    else:
        raise

pid = os.fork()
if pid == 0: # child
    listen_socket.close() # close child copy
    handle_request(client_connection)
    client_connection.close()
    os._exit(0)
else: # parent
    client_connection.close() # close parent copy and loop over

if __name__ == '__main__':
    serve_forever()
```

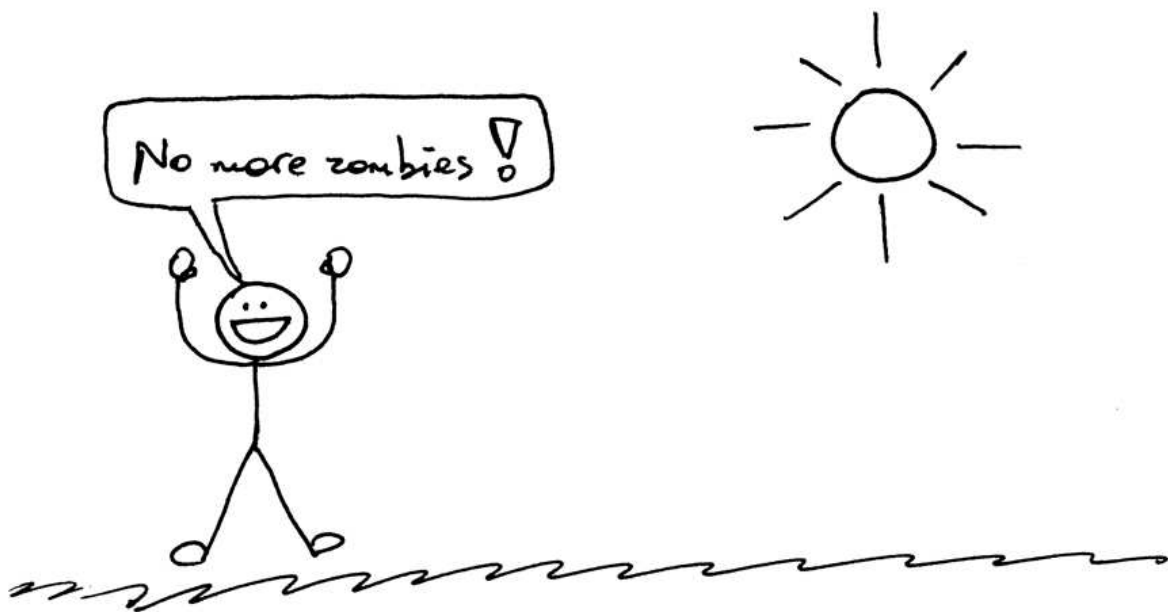
Start the server:

```
$ python webserver3g.py
```

Use the test client `client3.py` (<https://github.com/rspivak/lbaws/blob/master/part3/client3.py>):

```
$ python client3.py --max-clients 128
```

And now verify that there are no more zombies. Yay! Life is good without zombies :)



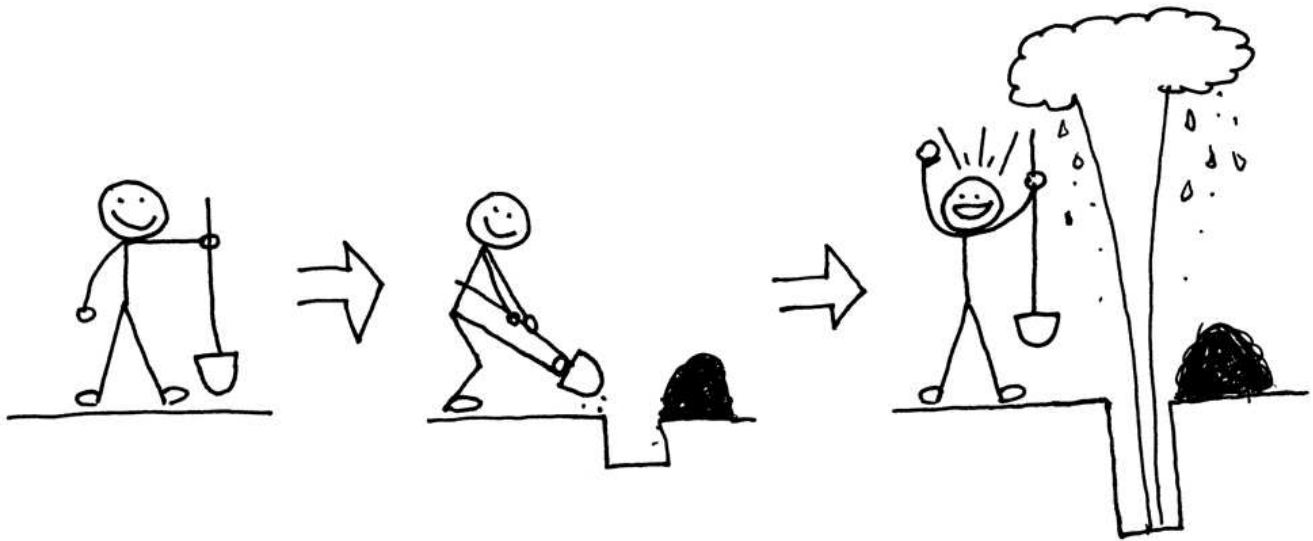
Congratulations! It's been a pretty long journey but I hope you liked it. Now you have your own simple concurrent server and the code can serve as a foundation for your further work towards a production grade Web server.

I'll leave it as an exercise for you to update the WSGI server from [Part 2](#) (<http://ruslanspivak.com/lbaws-part2/>) and make it concurrent. You can find the modified version [here](https://github.com/rspivak/lbaws/blob/master/part3/webserver3h.py) (<https://github.com/rspivak/lbaws/blob/master/part3/webserver3h.py>). But look at my code only after you've implemented your own version. You have all the necessary information to do that. So go and just do it :)

What's next? As Josh Billings said,

“Be like a postage stamp — stick to one thing until you get there.”

Start mastering the basics. Question what you already know. And always dig deeper.



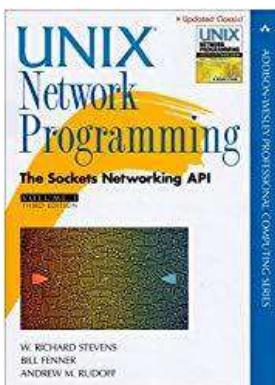
“If you learn only methods, you’ll be tied to your methods. But if you learn principles, you can devise your own methods.” —Ralph Waldo Emerson

Below is a list of books that I’ve drawn on for most of the material in this article. They will help you broaden and deepen your knowledge about the topics I’ve covered. I highly recommend you to get those books somehow: borrow them from your friends, check them out from your local library, or just buy them on Amazon. They are the keepers:

1. Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition)

([http://www.amazon.com/gp/product/0131411551/ref=as_li_tl?](http://www.amazon.com/gp/product/0131411551/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0131411551&linkCode=as2&tag=russblo0b-20&linkId=2F4NYRBND566JJQL)

[ie=UTF8&camp=1789&creative=9325&creativeASIN=0131411551&linkCode=as2&tag=russblo0b-20&linkId=2F4NYRBND566JJQL](http://www.amazon.com/gp/product/0131411551/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0131411551&linkCode=as2&tag=russblo0b-20&linkId=2F4NYRBND566JJQL))



([http://www.amazon.com/gp/product/0131411551/ref=as_li_tl?](http://www.amazon.com/gp/product/0131411551/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0131411551&linkCode=as2&tag=russblo0b-20&linkId=V6X5YJJ7PJZEY2QG)

[ie=UTF8&camp=1789&creative=9325&creativeASIN=0131411551&linkCode=as2&tag=russblo0b-20&linkId=V6X5YJJ7PJZEY2QG](http://www.amazon.com/gp/product/0131411551/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0131411551&linkCode=as2&tag=russblo0b-20&linkId=V6X5YJJ7PJZEY2QG))

2. Advanced Programming in the UNIX Environment, 3rd Edition

([http://www.amazon.com/gp/product/0321637739/ref=as_li_tl?](http://www.amazon.com/gp/product/0321637739/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321637739&linkCode=as2&tag=russblo0b-20&linkId=3ZYAKB537G6TM22J)

[ie=UTF8&camp=1789&creative=9325&creativeASIN=0321637739&linkCode=as2&tag=russblo0b-20&linkId=3ZYAKB537G6TM22J](http://www.amazon.com/gp/product/0321637739/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321637739&linkCode=as2&tag=russblo0b-20&linkId=3ZYAKB537G6TM22J))